Encapsulation and Behavioral Inheritance in a Synchronous Model of Computation for Embedded System Services Adaptation

Mickael Kerboeuf, Jean-Pierre Talpin

▶ **To cite this version:**

Mickael Kerboeuf, Jean-Pierre Talpin. Encapsulation and Behavioral Inheritance in a Synchronous Model of Computation for Embedded System Services Adaptation. Journal of Logic and Algebraic Programming, 2005, 63 (2), pp.241-269. 10.1016/j.jlap.2004.05.005 . hal-00546408

HAL Id: hal-00546408
https://hal.science/hal-00546408

Submitted on 14 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Encapsulation and Behavioral Inheritance in a Synchronous Model of Computation for Embedded System Services Adaptation

Mickaël Kerbœuf [a,*] Jean-Pierre Talpin [a]

[a] IRISA, *Campus de Beaulieu, 35042 Rennes cedex, France*

Abstract

Because it encourages the incremental development of software and the reuse of code by abstracting away implementation details, object orientation is an intuitive and sensible way to conceive large software out of existing application components and libraries. In practice, however, object-orientation is most of the time applied and used with sequentiality in mind. This practice may sometimes be conceptually inadequate for, e.g., control-dominated reactive system components.

We address this issue by proposing a process calculus that melts the paradigm of synchronous programming to key object-oriented features: *encapsulation* and *behavioral inheritance* with overriding by means of specific *algebraic concurrency combinators*. This framework provides support for the reuse of components and, more specifically, for the adaptation of embedded systems with new services.

Cast in the context of a strict interpretation of the synchronous hypothesis, the proposed model supports a static interpretation of inheritance: overriding is resolved at compile-time (or link-time) and inheritance combinators are translated into primitive synchronous ones. This compilation technique puts object-orientation to work in a syntax-oriented model of synchronous concurrency that naturally supports the incremental, refinement-based design of concurrent systems starting from encapsulated and reused application components.

The benefits of our approach are illustrated by a concrete and practical example: the adaptation of services to a plain old telephone service specification.

Key words: synchronous programming, process calculi, encapsulation, reuse and behavioral inheritance.

* Corresponding author.
Email addresses: Mickael.Kerboeuf@irisa.fr (Mickaël Kerbœuf), Jean-PierreTalpin@irisa.fr (Jean-Pierre Talpin).
URLs: http://www.irisa.fr/prive/kerboeuf (Mickaël Kerbœuf),

1 Introduction

Object-orientation favors an incremental development of sequential software by taking into account the structural and behavioral refinement of program components using the concept of inheritance. Object-orientation enables the reuse of program libraries by abstracting away implementation details from the necessary information contained in an interface, a signature, a type. A few object-oriented concepts, materialized by a small set of operators, with a clear and formal semantics, provide effective solutions for the design of large sequential software.

Moving to the design of concurrent systems, however, the picture is not that satisfactory. It is indeed a very challenging issue to give an object-oriented account to concurrency that meets the same degree of simplicity as for sequential software.

The synchronous hypothesis is an efficient approach to the design of concurrent and control-dominated software. Synchrony consists of assuming that communications and computations are instantaneous during the successive execution steps of a system. Making this hypothesis is beneficial to system design. It allows the designer to focus on the logics of the system, characterized by synchronization and causal relations between events, and abstract away timing issues until a latter stage of system design (until its mapping on a given architecture).

We propose a new calculus of synchronous processes that supports the incremental, object-oriented design of synchronous system components. This model consists of a core algebraic formalism, akin to Pnueli's synchronous transition systems [19], that melts the paradigm of synchronous programming to the notions of *encapsulation* and of *inheritance* with *overriding*, borrowed to object-oriented programming. The classical notion of class is introduced as an abstract parameterized and encapsulated process. An object is an instance of a class. An inheritance operator is defined at the class level. It refines the behavior of an initial class with a special class that corresponds to the notion of wrapper. A concurrent behavioral inheritance operator is defined in terms of synchronous composition by introducing a technique of renaming or rewriting.

This intuitive and syntax-oriented approach offers flexible implementation possibilities: it can both been used to interpret behavioral inheritance in the context of a functional architecture consisting of process signatures and components, it can be used to combine, compile and optimize concurrent objects, it can be used to link and load separately compiled modules. In conclusion, it fully supports incremental design, reuse and encapsulation of objects for the component-based engineering of concurrent software.

Overview In section 2, we first give a brief overview of the synchronous paradigm before introducing the core algebraic model of implicit synchronous transition systems (ISTS). Section 3 gives the syntax and semantics of object orientation of this model: it is extended with

`http://www.irisa.fr/prive/talpin` (Jean-Pierre Talpin).

a mechanism for encapsulation, and with synchronous behavioral inheritance. A technique for performing a static resolution of behavioral inheritance is then presented. The benefits of this approach are probed and illustrated in section 4 by considering the concrete and practical example of the adaptation of services of a POTS: a plain old telephone service.

2 A synchronous approach for the design of reactive systems

2.1 Synchrony and asynchrony

Synchrony and asynchrony are fundamentally different concepts in nature. Asynchrony is traditionally relevant for reasoning on distributed algorithms and for modeling non-determinism, failure, mobility. It meets a natural implementation by networked point-to-point communication. Synchrony is more commonly viewed as specific to the design of reactive systems and digital circuits. In this context, timeless logical concurrency and determinism are suitable hypotheses.

Time prevails in an asynchronous design as communication and computation times need to be taken into account at every level of the system under design. The absence of a common reference of time requires one to manage the *local* execution context of each application component and maintain the expected *global* behavior of the system. In this process, non-determinism incurred by asynchronous interactions increases the number of possible sequences of interleaved events. This makes the proof of suitable invariants (safety properties, absence of live-locks or dead-locks) harder.

By contrast, a synchronous design hypothesis consists of assuming that communications and computations are instantaneous between the successive execution steps of a system. Making this hypothesis is beneficial for design. It allows the designer to focus on the logics of the system, characterized by synchronization and causal relations between events, and abstract away timing issues until a later stage of the design (its deployment on a given architecture).

In the synchronous approach of concurrency, time is *abstracted* from computations and communications. As computing takes no time, the behavior of a computational unit can be seen as a sequence of simultaneous events, ordered by causal relations. As communication takes no time, a message from a unit to another is sent and received at the same logical instant. Thus, synchrony offers a global view of the interaction in a system where the only notions relevant to verification are simultaneity of events or causal precedence between events.

Back to the real world, where physical time has to be taken into account, the synchronous hypothesis can be validated by checking that the program reacts rapidly enough to perceive the events in suitable order, or by checking the so-called property of *endochrony*. This property expresses that a *unique* sequence of interleaving events can be inferred from the synchronous specification, regardless of the delay induced by each event.

2.2 Synchronous languages

Synchrony imposes a discretization of the behavior of a system: a synchronous specification is a sequence of instantaneous reactions to events. Imperative synchronous languages like STATECHARTS [13] or ESTEREL [4] focus on the sequence of events. Declarative (data-flow) synchronous languages such as LUSTRE [12] or SIGNAL [3] focus on the elementary reactions: a program handles streams of values, the *signals*. During a given execution step, each signal is either "present" or "absent" (but a computation is never "in progress").

Building upon previous work on casting the synchronous multi-clocked model of computation of SIGNAL into notions of process calculi [21], we define an algebraic model of implicit synchronous transition system, which we call ISTS, akin to Pnueli's synchronous transition systems (STS, [19]), where absence is explicit (for verification purposes) and to the SIGNAL modeling language, where absence is implicit.

2.3 A calculus of synchronous processes

The ISTS formalism aims at supporting the introduction of new concurrency concepts to ease the compositional modeling of reactive systems starting from a minimal set of constructs.

ISTS borrows an operator of non-deterministic choice between behaviors from STS [19] in order to support a structural equivalence relation which enables syntax-oriented behavioral reasoning on processes. ISTS differs from STS by letting absence be an implicit (non syntactic) notion in the model (as in SIGNAL). The equivalence between SIGNAL and the ISTS is shown in [14].

In the remainder of this section, we first give an informal overview of the ISTS centered around an example. Then, its formal syntax and operational semantics are detailed. They are summarized in appendix A.

2.3.1 Overview of the ISTS *formalism*

In ISTS, a synchronous *process* consists of a set of relations or partial equations on *signals*. A signal is identified by a *name* x which, at any logical instant of time (each transition), either carries a value v (and then we say that this signal is *present*), or not (and we say that it is *absent*, making use of the special mark $\perp$ to denote this absence). The *clock* of a signal x (denoted by $\hat{x}$) is the set of instants when this signal is present.

Each elementary equation or transition relation of a process specifies a relation between the values of its input and output signals.

For instance, $(z{=}x+y)$ is a primitive addition process. It relates the the integer *input* signals

x and y to the integer *output* signal z. The *values* carried by x, y and z, but also their *clock* $\hat{x}$, $\hat{y}$ and $\hat{z}$, are related: x, y and z are present at the same time, and when x and y carry the values c and d, then z carries the value $c+d$. Usual operators on numbers and booleans are provided. *Identity* (or assignment) is simply written $(y{=}x)$.

Guards are primitive processes. They enable to trigger reactions or transitions under certain conditions. However, they do not define any output signal. For instance, the guard $(\mathsf{when}\ x)$ (resp. $(\mathsf{when\ not}(x))$) is active only if the boolean input signal x carries the value *true* (resp. *false*). The guard $(\mathsf{event}\ x)$ is less restrictive. It is active only when the input signal x is present.

The *silent* process, denoted by $\mathsf{1}$, enables stuttering: it is active when all signals are absent. State transitions are implemented by the primitive process $(y{=}(\mathsf{pre}\, c)\, x)$. The function $(\mathsf{pre}\, c)$ defines a register which initially contains the value c. When the input signal x is present with the value d, the value c is sent along the output signal y, the value d is stored in the register and the process becomes $(y{=}(\mathsf{pre}\, d)\, x)$.

The synchronous composition of two processes p and q is written $p \mid q$. The transition of $p \mid q$ is performed by the simultaneous transition of p and q and with the same context (by context, it is meant that, if p assumes any signal x present with a value v or absent, then q should simultaneously make the same assumption during its transition). Non-deterministic choice is written $p{+}q$. It consists of choosing to execute either p or q during a given transition. Restriction p/x is used to limit the scope of a signal x to the process p.

Example 1 *Let* balance *be a process that implements a voting balance counter,* i.e. *it counts the number of times a signal x is true* minus *the number of times it is false. The* balance *is written:*

$$\mathsf{balance} \stackrel{\mathsf{def}}{=} 1 + \left((m{=}(\mathsf{pre}\, 0)\, n) \;\middle|\; \left(\begin{array}{r} (\mathsf{when}\, x) \mid (n{=}m{+}1) \\ +\ (\mathsf{when}\, (\mathsf{not}\, x)) \mid (n{=}m{-}1) \end{array} \right) \right) /m$$

At the top-level, the balance *consists of a choice between a process activated when x is present, and the silent process, to enable stuttering. If x is present, there are two possible* transitions.

The first transition is triggered if (and only if) x is true (it is guarded by $\mathsf{when}\, x$*). Simultaneously (*i.e. *by synchronous composition), m takes the previous value of n (initially 0) and n takes the value of $m+1$. The second transition is triggered iff x is false (it is guarded by* $\mathsf{when}\, (\mathsf{not}\, x)$*). If so, the balance count n is decremented.*

The balance *receives the input x and defines the output n. The signal m is used to calculate the current value of n given its previous one. It is defined locally. A possible sequence of values of the signals x, m and n in time can be depicted by considering the following possible*

trace of the execution of balance*:*

input: x	ff	ff	tt	$\bot$	ff	tt	tt	tt	$\ldots$
local : m	0	-1	-2	$\bot$	-1	-2	-1	0	$\ldots$
output: n	-1	-2	-1	$\bot$	-2	-1	0	1	$\ldots$

When x is false (value ff), the reaction guarded by when (not x) *is triggered. When x is true (value tt), the reaction guarded by* when x *is triggered. When x is absent (mark $\bot$), the silent reaction guarded by* 1 *is triggered. Notice that the signals m, n and x are* synchronous*: they are all absent or present at the same time.*

2.3.2 Formal syntax

We now introduce the syntax of Ists more formally. To do so, a few notational conventions used along the article are in order. Let $\mathcal{A}$ be a set and $a \in \mathcal{A}$. We write $\mathcal{A}^k$ for the set of sequences of length $k \in \mathbb{N}$ of elements of $\mathcal{A}$. We write $\mathcal{A}^*$ for $\bigcup_{k\in\mathbb{N}} \left(\mathcal{A}^k\right)$. A sequence of any length is denoted by $\tilde{a} \in \mathcal{A}^*$, and we write $(a_1, \ldots, a_k) \in \mathcal{A}^k$ for a sequence of length k.

We write $\mathbb{Z}$ and $\mathbb{B} = \{\mathit{tt}, \mathit{ff}\}$ for the domains of integers and booleans and $\mathcal{C} = \mathbb{B} + \mathbb{Z}$ for the set of constants. We consider an infinite countable sets of signals $x, y \in \mathcal{X}$ and functions $f, g \in \mathcal{F}$ (we assume $\mathcal{X}$ and $\mathcal{F}$ disjoint: $\mathcal{X} \cap \mathcal{F} = \emptyset$).

A process p in Ists consists of elementary transitions $(y{=}(\mathsf{pre}\, c)x)$ and simultaneous equations on signal names $(\tilde{y}{=}f\tilde{x})$ combined using synchronous composition $p \,|\, q$ and non-deterministic choice $p{+}q$. The sequences $\tilde{y}$ and $\tilde{x}$ of signals required and defined in an equation $(\tilde{y}{=}f\tilde{x})$ can be empty (to capture guards, constants and silence), and the empty sequence is denoted by $()$. Restriction p/x is used to limit the scope of a signal x to the process p.

$$
\begin{array}{rlll}
p, q ::= & \tilde{y}{=}f\tilde{x} & \text{(equation)} & \qquad c, d \in \mathcal{C} = \mathbb{B} + \mathbb{Z} \quad \text{(constant)} \\
\mid & y{=}(\mathsf{pre}\, c)x & \text{(transition)} & \qquad f, g \in \mathcal{F} \quad \text{(function)} \\
\mid & p \,|\, q & \text{(composition)} & \qquad x, y \in \mathcal{X} \quad \text{(signal)} \\
\mid & p{+}q & \text{(choice)} & \qquad (x_1, \ldots, x_k) \in \mathcal{X}^k \quad \text{(sequence)} \\
\mid & p/x & \text{(restriction)} & \qquad \tilde{x} \in \mathcal{X}^* = \bigcup_{k\in\mathbb{N}} \left(\mathcal{X}^k\right)
\end{array}
$$

2.3.3 Operational semantics of synchronous processes

The operational semantics of Ists consists of a set of axioms and rules that define the possible transition of a process by induction on its syntax. We first introduce the algebraic laws of Ists.

We write $\mathrm{fv}(p)$ and $\mathrm{dv}(p)$ for the set of free and defined names of a process p. Informally, a name x is free (resp. defined) in p if it occurs unbound in an action (resp. unbound and is an *output* signal of a base process) of p. We write $p[x/y]$ for the substitution of y by x in p and $\mathrm{dom}\, S$ for the domain of a substitution S.

$$\mathrm{fv}(\tilde{y} = f\tilde{x}) = \tilde{y} \cup \tilde{x} \qquad \mathrm{fv}(p{+}q) = \mathrm{fv}(p \,|\, q) = \mathrm{fv}(p) \cup \mathrm{fv}(q) \qquad \mathrm{fv}(p/x) = \mathrm{fv}(p) \setminus \{x\}$$
$$\mathrm{dv}(\tilde{y} = f\tilde{x}) = \tilde{y} \qquad \mathrm{dv}(p{+}q) = \mathrm{dv}(p \,|\, q) = \mathrm{dv}(p) \cup \mathrm{dv}(q) \qquad \mathrm{dv}(p/x) = \mathrm{dv}(p) \setminus \{x\}$$

Let $\mathcal{P}$ be the set of ISTS processes. The structural or syntactic equivalence relation $\equiv$ is defined on $\mathcal{P}$ (relations that involve scoping are subject to the side-condition $(*) : x \notin \mathrm{fv}(p)$).

$$p/y{\equiv}(p[x/y])/x^{(*)} \qquad p \,|\, (q \,|\, r){\equiv}(p \,|\, q) \,|\, r \qquad p{+}q{\equiv}q{+}p \qquad p/x{\equiv}p^{(*)}$$
$$p/x/y{\equiv}p/y/x \qquad p{+}(q{+}r){\equiv}(p{+}q){+}r \qquad p \,|\, q{\equiv}q \,|\, p$$
$$p \,|\, q/x{\equiv}(p \,|\, q)/x^{(*)} \qquad p{+}q/x{\equiv}(p{+}q)/x^{(*)} \qquad p \,|\, \mathsf{1}{\equiv}p{+}p \equiv p$$

The operational semantics of a process p is defined by the relation $p \xrightarrow{\boldsymbol{e}} q$. It defines the possible transitions $\boldsymbol{e}$ of a process from an initial state p to a final state q. The term $\boldsymbol{e}$ represents the events that are present in the environment of the process at the instant at which the transition takes place. It is constructed by induction on the term p by combining events from every sub-term of p. An event is defined by the association of a signal x to a value c in $\boldsymbol{e}$, written $x \mapsto c$. It denotes the value c carried by the signal x at the (logical) instant denoted by $\boldsymbol{e}$. A signal x can alternatively be regarded as absent (i.e. x is absent iff $x \notin \mathrm{dom}\, \boldsymbol{e}$).

$$\boldsymbol{e}, \boldsymbol{f} \in \mathcal{E} = \mathcal{X} \rightharpoonup \mathcal{C}^* \quad \text{(environment)}$$

Rule (eqv) takes into account the syntactic recombination of processes. **Rule (or)** is the choice rule. It allows a transition from $p{+}q$ to $r{+}q$ with $\boldsymbol{e}$ if a transition from p to r with $\boldsymbol{e}$ is possible (resp. from q, by rule (eqv)). **Rule (let)** implements the scope restriction of a name x in a process p. We write $\boldsymbol{e}_x$ for the context $\boldsymbol{e}$ outside of the scope of x ($x \notin \mathrm{dom}\, \boldsymbol{e}_x$, for all x).

$$(\text{eqv})\ \frac{p \equiv p' \xrightarrow{\boldsymbol{e}} q' \equiv q}{p \xrightarrow{\boldsymbol{e}} q} \qquad (\text{or})\ \frac{p \xrightarrow{\boldsymbol{e}} r}{p{+}q \xrightarrow{\boldsymbol{e}} r{+}q} \qquad (\text{let})\ \frac{p \xrightarrow{\boldsymbol{e}} q}{p/x \xrightarrow{\boldsymbol{e}_x} q/x}$$

Rule (and) implements synchronous composition. It stipulates that the simultaneous transitions from p to p' with $\boldsymbol{e}$ and from q to q' with $\boldsymbol{f}$ are valid iff $\boldsymbol{e}$ and $\boldsymbol{f}$ agree on the assignment to all signals shared by p and q, as defined by the side-condition. More precisely, a signal x shared by p and q ($x \in \mathrm{fv}(p) \cap \mathrm{fv}(q)$) must be simultaneously present or absent in both p and q ($x \in \mathrm{dom}\, \boldsymbol{e} \Leftrightarrow x \in \mathrm{dom}\, \boldsymbol{f}$) and, when present ($x \in \mathrm{dom}\, \boldsymbol{e} \cap \mathrm{dom}\, \boldsymbol{f}$), with the

same value ($\boldsymbol{e}(x) = \boldsymbol{f}(x)$).

$$(\text{and})\ \frac{p \xrightarrow{\boldsymbol{e}} p' \qquad q \xrightarrow{\boldsymbol{f}} q'}{p \,|\, q \xrightarrow{\boldsymbol{e} \cup \boldsymbol{f}} p' \,|\, q'} \qquad \text{iff } \forall x \in \mathrm{fv}(p) \cap \mathrm{fv}(q),\ \begin{cases} (x \in \mathrm{dom}\,\boldsymbol{e} \Leftrightarrow x \in \mathrm{dom}\,\boldsymbol{f}) \\ \wedge\ (x \in \mathrm{dom}\,\boldsymbol{e} \cap \mathrm{dom}\,\boldsymbol{f} \Rightarrow \boldsymbol{e}(x) = \boldsymbol{f}(x)) \end{cases}$$

Axiom (com) defines the meaning of primitive functions (and, in extenso constants). At a given transition, an equation ($\tilde{y}{=}f\tilde{x}$) relates the *values* $\tilde{d} \in \mathcal{C}^*$ carried by the sequence of input signals $\tilde{x}$ to the values $\tilde{c} \in \mathcal{C}^*$ carried by the sequence of output signals $\tilde{y}$ according to a (possibly partial) function f. A partial map δ defines how primitive functions (e.g. identity, equality or boolean and integer functions) relates these values:

$$\delta :\ \mathcal{F} \rightharpoonup (\mathcal{C}^* \rightharpoonup \mathcal{C}^*)$$
$$\forall f \in \mathrm{dom}\,\delta, \quad \exists k, k' \text{ s.t.} \quad \delta(f) : \mathcal{C}^k \rightharpoonup \mathcal{C}^{k'}$$

If a function f does not return any output value (i.e. $\delta(f) : \mathcal{C}^k \rightharpoonup \mathcal{C}^0$), f implements a *guard*, such as when and event. In this case, we write ($\mathsf{when}\,x$) for ($()$=$\mathsf{when}\,x$). If f does not require any input value (i.e. $\delta(f) : \mathcal{C}^0 \rightharpoonup \mathcal{C}^{k'}$), f implements a constant signal, like true and false, which stands for the boolean constants tt and ff. In this case, we write (x=true) or (x=tt) for (x=$\mathsf{true}\,()$). The silent process $\mathsf{1}$ is defined by the function which neither defines any output signal nor requires any input signal (i.e. $\delta(f) : \mathcal{C}^0 \rightharpoonup \mathcal{C}^0$). We just write $\mathsf{1}$ instead of ($()$=$\mathsf{1}\,()$).

$$\begin{aligned}
\delta(+) &= \{((c,c') \mapsto (d)) \,|\, (c,c',d) \in \mathbb{Z}^3 \wedge d{=}c{+}c'\} & \delta(\mathsf{when}) &= \{((tt) \mapsto ())\} \\
\delta(\mathsf{id}) &= \{((c) \mapsto (c)) | c \in \mathcal{C}\} & \delta(\mathsf{true}) &= \{(() \mapsto (tt))\} \\
\delta(=) &= \{((c,c) \mapsto (tt)) | c \in \mathcal{C}\} \cup \{((c,c') \mapsto (ff)) | c \neq c'\} & \delta(\mathsf{false}) &= \{(() \mapsto (ff))\} \\
\delta(\mathsf{event}) &= \{((c) \mapsto ()) \,|\, c \in \mathcal{C}\} & \delta(\mathsf{1}) &= \{(() \mapsto ())\}
\end{aligned}$$

The side condition of (com) makes the use of δ explicit. It stipulates that the transition across ($\tilde{y}{=}f\tilde{x}$) is possible with $\boldsymbol{e}$ iff $\boldsymbol{e}$ is defined for (and *only* for) $\tilde{y}$ and $\tilde{x}$, and if the values carried by $\tilde{y}$ and $\tilde{x}$ satisfy $\delta(f)$. Notice that $\boldsymbol{e} : \mathcal{X} \rightharpoonup \mathcal{C}^*$, and hence, we write $\boldsymbol{e}(\tilde{x})$ for the sequence $(\boldsymbol{e}(x_1), \ldots, \boldsymbol{e}(x_n))$ where $(x_1, \ldots, x_n) = \tilde{x}$.

$$(\text{com}) \qquad (\tilde{y}{=}f\tilde{x}) \xrightarrow{\boldsymbol{e}} (\tilde{y}{=}f\tilde{x}) \qquad \text{iff } \mathrm{dom}\,\boldsymbol{e} = \tilde{x} \cup \tilde{y} \ \wedge\ \delta(f)(\boldsymbol{e}(\tilde{x})) = \boldsymbol{e}(\tilde{y})$$

¿From the axiom (com), the transition of guards, constants and silence can easily be deduced.

$$(z{=}x{+}y) \xrightarrow{\substack{x \mapsto c, y \mapsto d \\ z \mapsto c+d}} (z{=}x{+}y) \qquad 1 \xrightarrow{\emptyset} 1$$

$$x{=}tt \xrightarrow{x \mapsto tt} x{=}tt \qquad x{=}ff \xrightarrow{x \mapsto ff} x{=}ff$$

$$\mathsf{when}\,x \xrightarrow{x \mapsto tt} \mathsf{when}\,x \qquad \mathsf{event}\,x \xrightarrow{x \mapsto c} \mathsf{event}\,x$$

Notice that ($\mathsf{when}\,x$) and (x=tt) have the same behavior. However, ($\mathsf{when}\,x$) only *uses* the signal x (it is a control structure) whereas (x=tt) *defines* the signal x (it is an assignment).

Example 2 *To manifest the preemption capability of choice and composition in the* ISTS *let us consider a choice expression where a signal x only appears in one of the alternatives: $p \equiv y{=}x + y{=}0$ and put the expression p in a context $p \,|\, q$ such that $q \equiv x{=}1$. By definition of the rule (or), either $y{=}x$ or $y{=}0$ react, assuming an environment $\boldsymbol{e}_1$ such that $\mathrm{dom}\,\boldsymbol{e}_1 = \{x, y\}$ and $\boldsymbol{e}_1(y) = \boldsymbol{e}_1(x)$ or producing an environment $\boldsymbol{e}_2$ such that $\mathrm{dom}\,\boldsymbol{e}_2 = \{y\}$ and $\boldsymbol{e}_2(y) = 0$. By definition of the rule (com), q reacts by emitting the value 1 along x, producing $\mathrm{dom}\,\boldsymbol{f} = \{x\}$ and $\boldsymbol{f}(x) = 0$. Let us consider the possible combinations of these two expressions by the rule (and). We need to respect the side-condition of that rule, which stipulates that x, the signal shared by p and q, is present in $\boldsymbol{e}$ iff it is in $\boldsymbol{f}$. The only choice is $\boldsymbol{e} = \boldsymbol{e}_1$: the presence of x in the context q has preempted the reaction $y{=}0$ in the expression p. Had p been the expression $((y{=}x{+}1) + (y{=}x{-}1))$ then choice in the context q would have been non-deterministic, allowing either the left or right alternative to be fired.*

Axiom (pre) defines the transition that corresponds to evaluating $y{=}(\mathsf{pre}\,c)x$. The syntactic update performed in the axiom allows to load the initial value c in the output signal y and to simultaneously store the value d carried by the input signal x.

$$\text{(pre)} \qquad (y{=}(\mathsf{pre}\,c)x) \xrightarrow{x \mapsto d,\, y \mapsto c} (y{=}(\mathsf{pre}\,d)x)$$

A summary of the syntax and the operational semantics of ISTS is given in appendix A.

2.4 Related models

The synchronous interaction model in the ISTS is primarily related to synchronous formalisms. It essentially differs from related process calculi such as SCCS [17] in the role played by absence.

For instance, consider the SCCS process: $\bar{a} \times (a{+}\bar{b})$. If the event $\bar{a}$ occurs, the term on the right has the choice to fire a (and communicate), or to fire $\bar{b}$. In the ISTS, only the first transition is possible (for the same reason as for example 2): the action b can only be chosen if a (the other arm of the choice) is absent. This difference reflects the role of absence in synchronous formalisms. Another example is the process $\mathsf{balance}$: the transition by $\mathsf{1}$ is possible only if x is absent. Indeed, if x is absent, then $\mathsf{1}$ is the only term that can be triggered, since all other terms assume the presence of x. In other words, the absence of x is the triggering event for the silent transition $\mathsf{1}$.

3 Object-orientated aspects

In the previous section, we presented the ISTS formalism and the synchronous hypothesis on which it is founded. In this section, we define its encapsulated version, Objective Signal, and

then further augment it with a behavioral inheritance operator. To begin with, the principles of our synchronous object orientation are explained. Then, encapsulation and inheritance in Objective Signal are presented.

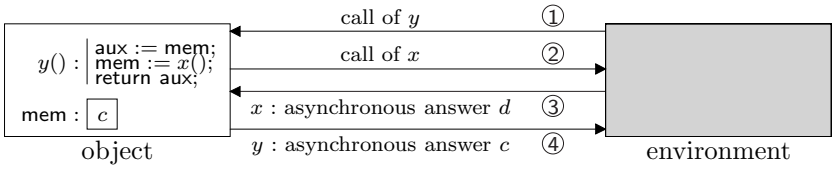
3.1 Motivations and principles

We essentially aim at defining formal methods enabling the reuse of objects and classes by employing an inheritance mechanism. This mechanism allows to adapt the behavior of a class or an object from the outside, without having to rewrite its implementation. Some important features of the object-oriented paradigm such as polymorphism, first-class objects and dynamic object creation are absent from our model. Their combination to a synchronous model of computation would raise issues, such as dynamic memory management, that are hardly compatible with the requirements of synchronous processes to execute within bounded (a priori predicted) space and time. We hence focus on the more fundamental merits of the object-oriented approach to provide means to favor the reusability of components and investigate the addition of encapsulation and inheritance mechanisms in a synchronous framework.

3.1.1 Objects and synchronous behaviors

An object is usually defined by a set of *methods* and *attributes*. Formally, an object is often represented by a record: an ordered and labeled collection of methods and attributes. The values associated to the attributes of an object represent its *state*. Methods enable to read and/or write the state of an object. The environment of an object is itself composed of several objects.

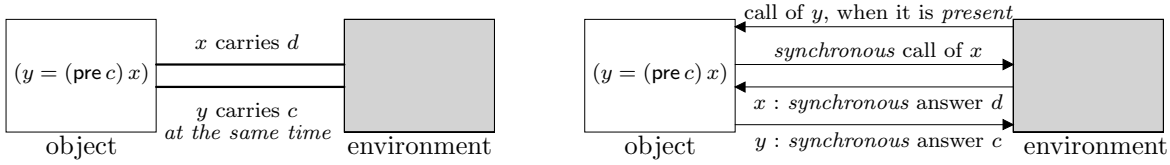
In conventional approaches to concurrent object-oriented programming, method calls are asynchronous. As an example, the next figure shows the interaction of an object, that defines the attribute mem and a method y, with its environment, which provides the method x. Calling y (step 1) triggers a call of x (step 2), performs an update of mem when x answers (step 3), and returns the last value of mem (step 4).



In a synchronous approach of concurrency, computation and communication take no time. From an external point of view, a process p of Objective Signal is characterized by *the signals it defines*. The state of p corresponds to the initial values c of delay equations $(y{=}(\mathsf{pre}\, c)\, x)$ occurring in p. The state of the object is modified iff a signal defined by such an equation is present.

The main departure of our framework from asynchronous models to concurrent object-orientation lies in the *synchronization relations* between an object and its environment: a defined signal can only be *called* (i.e. it is present and its value can be *fetched*) under some conditions and by some stimulus. The synchronization relations of an object are given by the *clocks* of the signals it involves. Remember that the clock of a signal denotes the instants at which the signal is present or triggered. Yet, notice that to conform with the synchronous paradigm, the call (i.e. the trigger) of a signal and its answer (i.e. its reaction) are *simultaneous* (computation and communication take no time).

The next figure illustrates the interaction between a process ($y{=}(\mathsf{pre}\, c)\, x$) and its environment. The picture on the right is a reformulation of the figure on the left which makes the analogy to the asynchronous interaction explicit.



The object-oriented representation of a synchronous process is a record that contains the signals it defines. At first glance, we need to add an encapsulation mechanism to that structure, in order to create abstract processes (classes). The defined signals of a synchronous process p are characterized by clocks (sets of triggering instants) and by the values they carry.

Next, we add inheritance (in the aim of supporting reuse and overriding). For the defined signals of an object, overriding implies the capability of modifying the result of the signal as well as its clock. Notice that synchronous composition and choice already (partially) achieve this requirement. Choice allows to extend a process with a new behavior (i.e. the definition and the clock of a signal) and synchronous composition allows to constrain a signal with new synchronization relations.

Still, we need to be able to reset *the values* carried by a signal. To this end, we introduce an *asymmetric* synchronous composition over classes, that enables to override the definition of a signal and to use its previous definition via the classical notion of super-variable.

3.1.2 Inheritance and static resolution

The Ists formalism, extended with encapsulation and inheritance, aims at melting object orientation, concurrency and synchronous data-flow within a same formal framework. However, perfect synchrony incurs strong specification requirements, notably on the resources of the system (memory size, computation time), which need to be *bounded*. For that reason, it is for instance not possible to recursively use a signal without introducing a *delay* between each recursive call or to dynamically allocate new resources at runtime. However, the topology of interaction being known statically enables to check precisely and efficiently the system safe.

Hence, in Objective Signal, the creation of an object (using a classical "new" class instantiation statement), builds the object from the model of the class and *activates* it. As a consequence, inheritance needs to be resolved statically, which may seem quite restrictive. However, our aim is less about defining a new model of execution (with resource allocation) and more on defining a suitable model for the *specification* of system behaviors using *encapsulation* and providing *reusability*.

3.1.3 Related works

The object-orientation of concurrent calculi has been a widely investigated topic. In [20], Pict, an object-oriented concurrent language founded on the π-calculus, is presented. It implements very powerful features like encapsulation, dynamicity and mobility, but its mechanism for reusability does not implement an inheritance mechanism taking into account compositional and structural modification of systems. In [9], a static and syntactic-oriented inheritance mechanism is defined for the Join-Calculus [10].

Objective Signal relates to that approach by adding a similar, syntax-oriented, mechanism to a synchronous formalism. Different approaches to reactive and synchronous concurrent objects have also been proposed in [5,6], which do not enables the overriding of the behaviors of objects.

Instead of adding object-oriented features to a synchronous or asynchronous concurrent formalism (such as the π-calculus or the ISTS), some further related works have investigated the extension of object-oriented formalisms with concurrency. Most of the approaches considered in this field are founded on the object-oriented calculus **imp**ς of Abadi and Cardelli [1], featuring classes, inheritance, prototyping, dynamic creations, subtyping and method specialization. In [11], **conc**ς is defined to encompass concurrent objects by including concurrency operators borrowed to the π-calculus. In [8], **conc**ς is further extended with a calculus of dependent types to analyze and avoid race conditions in concurrent specifications (i.e. the simultaneous access to the same ressource). The synchronous paradigm on which Objective Signal is founded does not aim at matching the expressive capability of **conc**ς yet casts encapsulation and behavioral inheritance in a (synchronous) framework were design errors such as race conditions can easily be analyzed and detected.

3.2 Encapsulation

3.2.1 Overview of encapsulation in Objective Signal

Processes p in Objective Signal are encapsulated within *classes* $\mathbb{C}$. A class gives the generic *definition* of an object that can be *instantiated* by providing its initial state. In the definition of a class, a *defined* signal x (i.e. which appears on the left hand-side of an equation) is *provided* by the class. A signal y, which appears on the right hand-side of an equation, is a

signal call or *signal fetch*. The class which *provides* this signal is identified by a parameter $C \in \mathcal{M}$. Thus, signal calls are prefixed by class parameters: $C.y$.

A class definition is parameterized by all the classes it uses. Among them, a special class parameter self refers to the current instance of the class.

Example 3 *The class* $\mathbb{C}_{\mathsf{balance}}$ *is an encapsulation of the process* balance. *The initial definition is encapsulated within a structure which specifies its* interface*:* $[C]$ *is the class parameter which provides the signal* x *used by the balance. The special parameter* self *refers to the class* $\mathbb{C}_{\mathsf{balance}}$ *itself (which provides the signals* n *and* m*). The scope of* m *is restricted to the definition of the class.* m *is a* private *signal.*

$$\mathbb{C}_{\mathsf{balance}} \stackrel{\text{def}}{=} [C]\,. \left[1 + \left((m{=}(\mathsf{pre}\,0)\,\mathsf{self}.n) \;\middle|\; \left(\begin{array}{r}(\mathsf{when}\,C.x)\,|\,(n{=}\mathsf{self}.m{+}1)\\ +\ (\mathsf{when}\,(\mathsf{not}\,C.x))\,|\,(n{=}\mathsf{self}.m{-}1)\end{array}\right)\right)/m\right]$$

The class $\mathbb{C}_{\mathsf{balance}}$ *can only be instantiated if it is given an effective parameter that provides the signal* x*. Objects are identified by names* $o \in \mathcal{O}$ *and are created by the construct* $o' = \mathsf{new}\ \mathbb{C}(\tilde{o})$ *where* $\mathbb{C}$ *is a class (*e.g. $\mathbb{C}_{\mathsf{balance}}$*) and* o' *the name of the instance. Thus,* o' *corresponds to the parameter* self*. The sequence* $\tilde{o}$ *provides the effective parameters required by* $\mathbb{C}$ *(*e.g. C*).*

Objects o, o' *in Objective Signal behave like processes in* Ists*. They are combined using synchronous composition* $o \,|\, o'$ *and choice* $o{+}o'$*. For instance, the following object is composed of two sub-objects. The first one (named* balance*) is an instance of the class* $\mathbb{C}_{\mathsf{balance}}$*. Its creation requires an object* env *which provides the signal* x*.* balance *and* env *are connected by synchronous composition.*

$$\mathbb{C}_{\mathsf{env}} \stackrel{\text{def}}{=} [].[1{+}(x{=}tt){+}(x{=}ff)]$$
$$\mathsf{balance} = \mathsf{new}\ \mathbb{C}_{\mathsf{balance}}(\mathsf{env}) \quad | \quad \mathsf{env} = \mathsf{new}\ \mathbb{C}_{\mathsf{env}}()$$

3.2.2 Formal syntax

We formally introduce the encapsulated Objective Signal. There are two grammars in this extended formalism. The first rule correspond to Ists processes p where *instantiated* behaviors can be specified. The second rule $\mathbb{C}$ corresponds to class definitions where *abstracted* behaviors can be specified.

The first rule extends the grammar of Ists with the class instantiation statement $o' = \mathsf{new}\ \mathbb{C}(\tilde{o})$ presented in the previous example. As the target of a signal call x is an object o,

signal names $x \in \mathcal{X}$ now appear as *instantiated paths* $o.x \in (\mathcal{O} \times \mathcal{X})$.

$$
\begin{array}{rcll}
p, q & ::= & p \,|\, q \mid p{+}q \mid p/m \mid \tilde{m}{=}f\tilde{m}' \mid m{=}(\mathsf{pre}c)m' \mid o' = \mathsf{new}\ \mathbb{C}(\tilde{o}) & \text{(instantiated process)} \\
m & ::= & o.x & \text{(instantiated path)} \\
o & \in & \mathcal{O} & \text{(object name)}
\end{array}
$$

Classes $\mathbb{C}$ consist of an interface where class parameters are declared, and an abstract behavior p_a is defined (for "abstract p"). In these abstract behaviors, signal names $x \in \mathcal{X}$ on the right hand-side of equations are replaced by *abstract paths* $C.x \in (\mathcal{M} \times \mathcal{X})$. An abstract path $C.x$ denotes the call to a signal x of the class parameter C. The path $\mathsf{self}.x$ refers to the signal x of the current class. A signal name x appearing on the left hand-side of an equation is a defined signal. It implicitly refers to the abstract path $\mathsf{self}.x$.

$$
\begin{array}{rcll}
\mathbb{C} & ::= & [\tilde{C}].[p_a] & \text{(class)} \\
p_a, q_a & ::= & p_a \,|\, q_a \mid p_a{+}q_a \mid p_a/x \mid \tilde{y}{=}f\tilde{n} \mid y{=}(\mathsf{pre}c)n & \text{(abstract process)} \\
n & ::= & C.x & \text{(abstract path)} \\
C & \in & \mathcal{M} \ni \mathsf{self} & \text{(class parameter)}
\end{array}
$$

The notations for the free and defined signals of processes ($\mathrm{fv}(p)$ and $\mathrm{dv}(p)$) are extended to objects and classes in order to take paths p_a into account. In the remainder, we exclusively consider classes that are well-formed, i.e. classes $[\tilde{C}].[p_a]$ such that $C.x \in \mathrm{fv}(p_a) \Rightarrow C \in \tilde{C}$.

Notice that, in the grammar p, some non-instantiated classes may coexist with instantiated ones. When a reaction containing an object declaration $o' = \mathsf{new}\ \mathbb{C}(\tilde{o})$ is triggered, a new instance of $\mathbb{C}$ (i.e. a process that corresponds to the definition p_a) is created. Names and paths of the original definition are substituted by the effective parameters $\tilde{o}$ and o'. In the following example, abstract paths are substituted by instantiated paths.

$$
o' = \mathsf{new}\left[\begin{array}{l}[C]\,. \\ \left(\begin{array}{l} (x{=}C.x{+}1) \\ \mid\ (y{=}\mathsf{self}.x{-}1) \end{array}\right)/x \end{array}\right]\left(\ o\ \right) \qquad \rightsquigarrow \qquad \left(\begin{array}{l} (o'.x{=}o.x{+}1) \\ \mid\ (o'.y{=}o'.x{-}1) \end{array}\right)/o'.x
$$

The renaming of signals and paths is achieved by a syntactic operator bind^o_σ which selectively applies a substitution σ to the class parameters that appear on the right hand-side of equations (signal calls), and changes each defined signal x into a path $o.x$. Here, the name o corresponds to the name of the current instance. We write $o.\tilde{y}$ for the tuple $(o.y_1, \ldots, o.y_n)$

where $(y_1, \ldots, y_n) = \tilde{y}$.

$$\mathsf{bind}^o_\sigma(p_a \mid q_a) = \mathsf{bind}^o_\sigma(p_a) \mid \mathsf{bind}^o_\sigma(q_a) \qquad \mathsf{bind}^o_\sigma(p_a/x) = (\mathsf{bind}^o_\sigma(p_a))/o.x$$
$$\mathsf{bind}^o_\sigma(p_a + q_a) = \mathsf{bind}^o_\sigma(p_a) + \mathsf{bind}^o_\sigma(q_a) \qquad \mathsf{bind}^o_\sigma(\tilde{y} = f\tilde{n}) = (o.\tilde{y} = f(\tilde{n}\sigma))$$

3.2.3 Operational semantics

We define the operational semantics of the encapsulated processes p. The core operational semantics of ISTS remains unchanged: we just need to additionally take instantiated paths $o.x$ into account (instead of simply signal names x). Thus, an environment $\boldsymbol{e}$ is now a partial function from *paths* to constants:

$$\boldsymbol{e}, \boldsymbol{f} \in \mathcal{E} = (\mathcal{O} \times \mathcal{X}) \rightharpoonup \mathcal{C}^* \quad \text{(environment)}$$

The creation of a class instance occurs at run time, when a reaction containing an object definition is triggered. We need to introduce a rule for that purpose. It simply applies the renaming mechanism on the definition p_a of a class. The name o' of the created object is substituted to the parameter self. The effective parameters $\tilde{o}$ are substituted to the class parameters $\tilde{C}$.

$$\text{(inst)} \quad \frac{\mathsf{bind}^{o'}_{[o'\tilde{o}/\mathsf{self}\tilde{C}]}(p_a) \xrightarrow{\boldsymbol{e}} p}{o' = \mathsf{new}\ [\tilde{C}].[p_a](\tilde{o}) \xrightarrow{\boldsymbol{e}} p}$$

3.3 Inheritance

3.3.1 Overview of Objective Signal

We now complete the definition of Objective Signal with an inheritance operator over classes. Using the notion of *wrapper* class, this construct allows to *synchronously* add new signals to a class and to *refine* or adapt existing ones, compositionally. When a signal defined in the wrapper is also defined in the class it is applied to, the new definition prevails. Still, it is possible to refer to the initial one thanks to the super-class: the parameter super.

Example 4 *We wish to modify the class* $\mathbb{C}_{\mathsf{balance}}$ *in order to incorporate a reset signal* r *(provided by the class that defines* x*). The signal* r *can be invoked only if* x *is present. It is used to reset the balance to* 0*. The state of* $\mathbb{C}_{\mathsf{balance}}$ *is managed by a* private *signal* m*. Implementing this upgrade without inheritance would break encapsulation. Using inheritance,*

it amounts to adding a wrapper *(on the right hand-side) to the initial class as follows:*

$$\mathbb{C}_{\mathsf{resettable\_balance}} \stackrel{\mathsf{def}}{=} [C]\,. \left[1 + \left((m{=}(\mathsf{pre}\,0)\,\mathsf{self}.n) \;\middle|\; \begin{pmatrix} (\mathsf{when}\,C.x)\,|\,(n{=}\mathsf{self}.m{+}1) \\ +\ (\mathsf{when}\,(\mathsf{not}\,C.x))\,|\,(n{=}\mathsf{self}.m{-}1) \end{pmatrix}\right)/m\right] \;\&\; [C']\,.\,[]\,. \left[1 + (n{=}\mathsf{super}.n) + \begin{pmatrix} (\mathsf{event}\ \mathsf{super}.n) \\ |\ (\mathsf{event}\ C'.r) \\ |\ (n{=}0) \end{pmatrix}\right]$$

As its syntax suggests, inheritance & is basically a sort of oriented (and non commutative) synchronous composition. There are two parts in the interface of a wrapper. The first part ($[C']$ in the example) enables to reuse *the parameters of the initial class ($[C]$ of $\mathbb{C}_{\mathsf{balance}}$). The second part possibly introduces new parameters ($[]$ in the example: no new parameters).*

Stuttering is still enabled by the wrapper (reaction 1*). The second reaction* $(n{=}\mathsf{super}.n)$ *is enabled if and only if* r *is absent: it can be triggered in a context that provides values for (and only for)* n *and* $\mathsf{super}.n$ *(recall example 2). In particular, the presence of* r *(referenced in the third reaction of the process) inhibits the reaction* $(n{=}\mathsf{super}.n)$*. Hence, if* r *is absent, the second reaction is triggered and the previous version of* n *prevails.*

The third reaction specifies that n *provides* 0 *instead of* $\mathsf{super}.n$ *when* r*, provided the class by* C' *(alias* C*) is present. We introduce a renaming scheme to merge* $\mathbb{C}_{\mathsf{balance}}$ *and its wrapper into a unique base class.* $\mathbb{C}_{\mathsf{resettable\_balance}}$ *is then equivalent to the following class:*

$$\mathbb{C}_{\mathsf{resettable\_balance}} \equiv [C]\,. \left[\begin{array}{l} 1 + \left((m{=}(\mathsf{pre}\,0)\,\mathsf{self}.n) \;\middle|\; \begin{pmatrix} (\mathsf{when}\,C.x)\,|\,(\boldsymbol{n'}{=}\mathsf{self}.m{+}1) \\ +\ (\mathsf{when}\,(\mathsf{not}\,C.x))\,|\,(\boldsymbol{n'}{=}\mathsf{self}.m{-}1) \end{pmatrix}\right)/m \\ \quad | \\ 1 + (n{=}\mathsf{self}.\boldsymbol{n'}) + \begin{pmatrix} (\mathsf{event}\ \mathsf{self}.\boldsymbol{n'}) \\ |\ (\mathsf{event}\ C.r) \\ |\ (n{=}0) \end{pmatrix} \end{array}\right]/\boldsymbol{n'}$$

Renaming in the initial class and the wrapper is selective*. The signal* n *is overridden. In order*

to enable the coexistence *of its initial definition and its new one within a same structure, n is changed into a new name $\boldsymbol{n'}$ (whose scope is restricted),* only where it is initially defined. *Where n is only* used *(in the definition of m), n remains unchanged.* Thus, the signal m still maintains a delayed version of (the new version of) n.

The reactions added by the wrapper are also adapted. The previous references to the "super" signal n are changed into references to the "self" signal $\boldsymbol{n'}$. Finally, the parameters C and C' are unified in the global structure under the name C. The signals x and r are now provided by the same class C.

Notice that it is not possible to extend the initial behavior just by adding a reaction outside the scope of m, because when n is reset to 0, m must register it. It is therefore not possible to achieve the modification without inheritance, or without breaking the encapsulation.

To create an instance of the modified class, we must extend its environment with the signal r. Consider the following process:

$$\mathbb{C}_{\mathsf{env}} \stackrel{\text{def}}{=} []\text{.}\Big[\ 1 + \Big((x{=}tt){+}(x{=}ff)\Big) \mid \Big((r{=}tt){+}1\Big)\ \Big]$$
$$\mathsf{resettable\_balance} = \mathsf{new}\ \mathbb{C}_{\mathsf{resettable\_balance}}(\mathsf{env}) \quad \mid \quad \mathsf{env} = \mathsf{new}\ \mathbb{C}_{\mathsf{env}}()$$

A possible sequence of values of the signals x, m, n and r in time is depicted by the following execution trace. Boxed values indicate which signals are taken into account in the definition of n. The signal r has a preemptive power on x. When it is present, the previous value of n is ignored and n is reset to 0.

input : x	*ff*	*ff*	*tt*	$\perp$	*ff*	*tt*	*tt*	*tt*	...
input : r	$\perp$	*tt*	$\perp$	$\perp$	$\perp$	*tt*	$\perp$	$\perp$	...
private : m	0	-1	0	$\perp$	1	0	0	1	...
private : $\boldsymbol{n'}$	-1	-2	1	$\perp$	0	1	1	2	...
public : n	-1	**0**	1	$\perp$	0	**0**	1	2	...

3.3.2 Formal syntax

The definition of a wrapper uses the parameters self and super. They respectively refer to the *whole modified* class and to the *initial* class modified by the wrapper. Among the other parameters of a wrapper, it is necessary to distinguish between the parameters that are already used in the initial class (first part of the interface), and the new parameters introduced by the wrapper (second part of the interface). In the previous example, the wrapper reuses the parameter C of balance. Thus, as expected, the signals x and r are provided by the same object. We could have specified a new parameter in the second part of

the interface in order to let the signals r and x be provided by potentially different classes.

$$[\tilde{C}_r].[\tilde{C}].[p_a] \qquad \text{(wrapper)}$$

Wrappers can be considered as a general form of class. Indeed, a base class can be seen as a wrapper which neither uses the parameters of the initial class ($\tilde{C}_r = ()$), nor the reference super in its definition:

$$[\tilde{C}].[p_a] \quad \stackrel{\text{def}}{=} \quad [].[\tilde{C}].[p_a] \text{ and } \forall x \in \mathcal{X},\ \mathsf{super}.x \notin \text{fv}(p_a) \qquad \text{(base class)}$$

The following syntax extends second level (classes) of the encapsulated ISTS with wrappers and inheritance:

$$\begin{aligned}
\mathbb{C} &::= [\tilde{C}_r].[\tilde{C}].[p_a] \quad | \quad \mathbb{C}\&\mathbb{C}' && \text{(class)}\\
p_a, q_a &::= p_a \,|\, q_a \mid p_a{+}q_a \mid p_a/x \mid \tilde{y}{=}f\tilde{n} \mid y{=}(\mathsf{pre}c)n && \text{(abstract process)}\\
n &::= C.x && \text{(abstract path)}\\
C &\in \mathcal{M} \supset \{\mathsf{super}, \mathsf{self}\} && \text{(class parameter)}
\end{aligned}$$

3.3.3 A spatial version of the method lookup algorithm

In classical object oriented programming languages like SmallTalk, the semantics of inheritance is given by the *method lookup algorithm*. When a method is called, its definition is looked up in the receiving class. If it is not found, the search starts again in its *super* class. When a method uses another method of the same class (reference to *this*) the search starts in the current class. When a method of the overridden class is called, the search start in the *super* class. In Objective Signal, a class $\mathbb{C}$ can be built incrementally using inheritance:

$$\mathbb{C} \quad \stackrel{\text{def}}{=} \quad \mathbb{C}_1\&\mathbb{C}_2\&\ldots\&\mathbb{C}_n$$

The call of a signal x initially (and only) defined in $\mathbb{C}_1$ triggers a *signal lookup* mechanism based on the same classical principle of object-oriented programming. However, thanks to the synchronous hypothesis, the signal lookup is synchronous to the call! Thus, the *temporal* iteration of the classical method-lookup algorithm is replaced by a *spatial* one. We introduce a syntactic operator called lookup which aims at deploying the classes $\mathbb{C}_1 \ldots \mathbb{C}_n$ in a unique base class.

Along the way, the defined signals and their different overridden versions coexist after a specific *renaming*. This renaming is achieved by using a substitution mechanism and a selective application operator $\mathcal{L}$. Let $\mathbb{C}_1$ be a base class and p_a its definition. Let $\mathbb{C}_2$ be a wrapper and q_a its definition. If a signal x of $\mathbb{C}_1$ (i.e. $x \in \text{dv}(p_a)$) is overridden in $\mathbb{C}_2$ (i.e. $x \in \text{dv}(q_a)$), the occurrences of x in p_a are replaced with a fresh name. This renaming is achieved by the

following substitution:

$$\beta_{p_a q_a} : \begin{cases} \operatorname{dom} \beta_{p_a q_a} = \operatorname{dv}(p_a) \cap \operatorname{dv}(q_a) \\ \forall x \in \operatorname{dom} \beta_{p_a q_a},\ x\beta_{p_a q_a} = x' \notin \operatorname{dv}(p_a) \cup \operatorname{dv}(q_a) \end{cases}$$

However, this substitution must be selectively applied. Indeed, the *calls* must remain unchanged, and now, they refer to the new version of the signal. For that purpose, we use the following operator:

$$\begin{aligned} \mathcal{L}_\beta\,(\,p_a \,|\, q_a\,) &= \mathcal{L}_\beta(p_a) \,|\, \mathcal{L}_\beta(q_a) & \qquad \mathcal{L}_\beta(p_a/x) &= \mathcal{L}_{\beta_x}\,(p_a)\,/x \\ \mathcal{L}_\beta\,(\,p_a{+}q_a\,) &= \mathcal{L}_\beta(p_a){+}\mathcal{L}_\beta(q_a) & \qquad \mathcal{L}_\beta(\tilde{y}{=}f\tilde{n}) &= \left((\tilde{y}\beta){=}f\tilde{n}\right) \end{aligned}$$

A wrapper can modify another wrapper. The syntactic operator lookup unifies the interfaces of two base classes (and/or wrappers) and applies $\mathcal{L}$ on them. In the result, the overridden (i.e. renamed) signals are restricted locally.

$$\begin{aligned} &\mathsf{lookup} \begin{pmatrix} \quad [\tilde{C_{r1}}]\ .\ [\tilde{C_1}]\ .\ [p_a] \\ \&\ [\tilde{C_{r2}}]\ .\ [\tilde{C_2}]\ .\ [q_a] \end{pmatrix} \\ &\quad = [\tilde{C_{r1}}]\ .\ [\tilde{C_1}\tilde{C_2}]\ . \\ &\qquad \left[\begin{pmatrix} \quad \mathcal{L}_{\beta_{p_a q_a}}(p_a) \\ \mid\ q_a[\tilde{C_{r1}}\tilde{C_1}/\tilde{C_{r2}}][\mathsf{self}.(x\beta_{p_a q_a})/\mathsf{super}.x] \end{pmatrix} /\operatorname{im} \beta_{p_a q_a} \right] \end{aligned}$$

The initial class (which can itself be a wrapper as well) uses the parameters $\tilde{C_1}$ and reuses the parameters $\tilde{C_{r1}}$. The last ones refer to the parameters of the class that the wrapper is supposed to modify. $\tilde{C_1}$ and $\tilde{C_{r1}}$ can be reused (via the alias $\tilde{C_{r2}}$) in the wrapper of the wrapper. In the result, $\tilde{C_{r1}}$ remains unchanged. The parameters $\tilde{C_1}$ of the initial class and the parameters $\tilde{C_2}$ of the wrapper refer to the classes used in the top-level class. Then, they both appear in the interface of the result. The initial definitions p_a and q_a are synchronously composed. $\mathcal{L}_{\beta_{p_a q_a}}$ is applied on p_a to rename overridden definitions but not their *calls*. In q_a, the parameters of the initial class $\tilde{C_{r1}}\tilde{C_1}$ replace the corresponding parameters $\tilde{C_{r2}}$ in the wrapper. The previous versions of overridden signals $\mathsf{super}.x$ are replaced by their new names $x\beta_{p_a q_a}$ in the global component, now referred to as self. All these new names ($\operatorname{im} \beta_{p_a q_a}$) are restricted to the definition of the new class.

Most of the time, the modified class is not a wrapper and its wrapper does not introduce any new parameter. In this case, $\tilde{C_{r1}} = ()$ and $\tilde{C_2} = ()$. The wrapping of this class naturally yields

to another *base class* with the same interface (as in the previous example $\mathbb{C}_{\mathsf{resettable\_balance}}$):

$$\begin{aligned}
&\mathsf{lookup}\left(\begin{array}{r} [\tilde{C_1}]\ .\ [p_a] \\ \&\ [\tilde{C_{r2}}]\ .\ [\ \ ]\ .\ [q_a] \end{array}\right) \\
&\quad = [\tilde{C_1}]\ . \\
&\qquad \left[\left(\begin{array}{l} \mathcal{L}_{\beta_{p_a q_a}}(p_a) \\ \mid\ q_a[\tilde{C_1}/\tilde{C_{r2}}][\mathsf{self}.(x\beta_{p_a q_a})/\mathsf{super}.x] \end{array}\right)/\mathrm{im}\,\beta_{p_a q_a}\right]
\end{aligned}$$

On more complex classes, the lookup operator is applied recursively. Base classes and base wrappers are kept unchanged by it.

$$\begin{aligned}
\mathsf{lookup}\,(\,\mathbb{C}\&\mathbb{C}'\,) &= \mathsf{lookup}\,(\,\mathsf{lookup}(\mathbb{C})\&\mathsf{lookup}(\mathbb{C}')\,) \\
\mathsf{lookup}(\mathbb{C}) &= \mathbb{C} \qquad \text{(with } \mathbb{C}\text{: base class)}
\end{aligned}$$

3.3.4 Operational semantics

To define the rule for the resolution of inheritance at runtime, we upgrade the rule (inst) by adding the renaming mechanism on the given class $\mathbb{C}$. The rule can be applied only if the result of lookup is a base class.

$$(\text{inst})\quad \frac{\mathsf{lookup}(\mathbb{C}) = [\tilde{C}].[p_a] \qquad\qquad \mathsf{bind}^{o'}_{[o'\tilde{o}/\mathsf{self}\tilde{C}]}(p_a) \xrightarrow{\boldsymbol{e}} p}{o' = \mathsf{new}\ \mathbb{C}(\tilde{o}) \xrightarrow{\boldsymbol{e}} p}$$

A summary of the syntax and the operational semantics of Objective Signal is given in appendix B

3.3.5 Static resolution of naming

The formalism upon which our object-oriented language is built is first-order: the object network's topology doesn't evolve during execution. Thus, it is possible to detect synchronization constraints between different system components *statically*. Hence, it is possible to detect and resolve inheritance statically. We introduce a compilation technique from a Objective Signal process p to a base ISTS process $[\![p]\!]$ that preserves behavioral equivalence. The translation consists of globally applying the operators lookup and bind on p and of performing a substitution $\sigma : (\mathcal{O} \times \mathcal{X}) \to \mathcal{X}$ in order to change instantiated paths into original signal

names: $\forall o.x \in (\mathcal{O} \times \mathcal{X}), \forall o'.x' \in (\mathcal{O} \times \mathcal{X})$, $(o \neq o') \vee (x \neq x') \Rightarrow (\sigma(o.x) \neq \sigma(o'.x'))$.

$$\begin{array}{rr} [\![o' = \mathsf{new}\ \mathbb{C}(\tilde{o})]\!] = [\![o' = \mathsf{new}\ \mathsf{lookup}(\mathbb{C})(\tilde{o})]\!] & [\![p \,|\, q]\!] = [\![p]\!] \mid [\![q]\!] \\ \left[\!\!\left[o' = \mathsf{new}\ [\tilde{C}].[p_a](\tilde{o})\right]\!\!\right] = \left[\!\!\left[\mathsf{bind}^{o'}_{[o'\tilde{o}/\mathsf{self}\tilde{C}]}(p_a)\right]\!\!\right] & [\![p{+}q]\!] = [\![p]\!] + [\![q]\!] \\ [\![\tilde{m} = f\tilde{m}']\!] = (\tilde{m}\sigma = f\tilde{m}'\sigma) & [\![p/m]\!] = [\![p]\!]\,/(m\sigma) \end{array}$$

As the same syntactic operators are used, we naturally obtain the following result:

Theorem 1

$$p \xrightarrow{\boldsymbol{e}} q \;\Leftrightarrow\; [\![p]\!] \xrightarrow{\boldsymbol{e}\sigma} [\![q]\!]$$

Proof sketch *The proof, detailed in [14], is by induction on the structure of synchronous systems. The rules of the* Ists *and its object-oriented version are quite similar,* modulo *renaming by* σ.

Let us consider the base case $p : \tilde{m} = f\tilde{m}'$ *and suppose that* $p \xrightarrow{\boldsymbol{e}} q$. *Rule* **(com)** *applies and requires* q *to have the form* $\tilde{m} = f\tilde{m}'$ *and* $\boldsymbol{e}$ *be such that* $\mathrm{dom}\,\boldsymbol{e} = \tilde{m} \cup \tilde{m}'$ *and* $\delta(f)(\boldsymbol{e}(\tilde{m}')) = \boldsymbol{e}(\tilde{m})$. *Let* $\sigma : (\mathcal{O} \times \mathcal{X}) \to \mathcal{X}$ *be the substitution specified by the translation scheme. We have that:*

$$\forall m, m',\ m \neq m' \Rightarrow m\sigma \neq m'\sigma$$

Using this substitution on instanciated paths, we obtain:

$$(\tilde{m}\sigma{=}f\tilde{m}'\sigma) \xrightarrow{\boldsymbol{e}\sigma} (\tilde{m}\sigma{=}f\tilde{m}'\sigma)$$

where $\mathrm{dom}\,\boldsymbol{e}\sigma = \tilde{m}\sigma \cup \tilde{m}'\sigma \subset \mathcal{X}$. *According to the translation scheme, we have:* $[\![p]\!] \xrightarrow{\boldsymbol{e}\sigma} [\![q]\!]$.

The case of the base process $(m = (\mathsf{pre}\ c)\ m')$ *is similar. In cases where processes are composed of several (structurally) simpler processes, we just need to invoke an induction hypothesis. We detail such induction steps for the case analysis of the synchronous composition* $(p \,|\, q)$ *and of the class instanciation* $(o' = \mathsf{new}\ \mathbb{C}(\tilde{o}))$

Consider the process $p \ : \ o' = \mathsf{new}\ \mathbb{C}(\tilde{o})$ *and suppose that* $p \xrightarrow{\boldsymbol{e}} q$. *Rule* **(inst)** *applies and initiates the following proof sequence:*

$$
\begin{array}{lll}
p \xrightarrow{\boldsymbol{e}} q \Leftrightarrow \mathsf{lookup}(\mathbb{C}) = [\tilde{C}].[p_a] \ \wedge \ \mathsf{bind}^{o'}_{[o'\tilde{o}/\mathsf{self}\tilde{C}]}(p_a) \xrightarrow{\boldsymbol{e}} q & & \textit{rule } \textbf{(inst)} \\
\Leftrightarrow \mathsf{lookup}(\mathbb{C}) = [\tilde{C}].[p_a] \ \wedge \ \left[\!\!\left[\mathsf{bind}^{o'}_{[o'\tilde{o}/\mathsf{self}\tilde{C}]}(p_a)\right]\!\!\right] \xrightarrow{\boldsymbol{e}\sigma} [\![q]\!] & & \textit{induction} \\
\Leftrightarrow \mathsf{lookup}(\mathbb{C}) = [\tilde{C}].[p_a] \ \wedge \ \left[\!\!\left[o' = \mathsf{new}\ [\tilde{C}].[p_a](\tilde{o})\right]\!\!\right] \xrightarrow{\boldsymbol{e}\sigma} [\![q]\!] & & \textit{translation scheme} \\
\Leftrightarrow [\![o' = \mathsf{new}\ \mathsf{lookup}(\mathbb{C})(\tilde{o})]\!] \xrightarrow{\boldsymbol{e}\sigma} [\![q]\!] & & \\
\Leftrightarrow [\![o' = \mathsf{new}\ \mathbb{C}(\tilde{o})]\!] \xrightarrow{\boldsymbol{e}\sigma} [\![q]\!] & & \textit{translation scheme} \\
\Leftrightarrow [\![p]\!] \xrightarrow{\boldsymbol{e}\sigma} [\![q]\!] & & \textit{definition of } p
\end{array}
$$

Consider the process $(p \,|\, q)$ *and suppose that* $(p \,|\, q) \xrightarrow{\boldsymbol{e}} p'$. *Rule* **(and)** *applies and provides the following proof sequence:*

$$
\begin{array}{lll}
(p \,|\, q) \xrightarrow{\boldsymbol{e}} p' \Leftrightarrow p \xrightarrow{\boldsymbol{e_1}} p_1 \ \wedge \ q \xrightarrow{\boldsymbol{e_2}} p_2 \ \wedge \ \boldsymbol{e} = \boldsymbol{e_1} \cup \boldsymbol{e_2} \ \wedge \ p' = (p_1 \,|\, p_2) \ \wedge & & \\
\qquad \forall m \in \mathrm{fv}(p) \cap \mathrm{fv}(q), \left| \begin{array}{l} (m \in \mathrm{dom}\, \boldsymbol{e_1} \Leftrightarrow m \in \mathrm{dom}\, \boldsymbol{e_2}) \ \wedge \\ (m \in \mathrm{dom}\, \boldsymbol{e_1} \cap \mathrm{dom}\, \boldsymbol{e_2} \Rightarrow \boldsymbol{e_1}(m) = \boldsymbol{e_2}(m)) \end{array} \right. & & \textit{rule } \textbf{(and)} \\
\Leftrightarrow [\![p]\!] \xrightarrow{\boldsymbol{e_1}\sigma} [\![p_1]\!] \ \wedge \ [\![q]\!] \xrightarrow{\boldsymbol{e_2}\sigma} [\![p_2]\!] \ \wedge \ \boldsymbol{e} = \boldsymbol{e_1} \cup \boldsymbol{e_2} \ \wedge \ p' = (p_1 \,|\, p_2) \ \wedge & & \\
\qquad \forall m \in \mathrm{fv}(p) \cap \mathrm{fv}(q), \left| \begin{array}{l} (m \in \mathrm{dom}\, \boldsymbol{e_1} \Leftrightarrow m \in \mathrm{dom}\, \boldsymbol{e_2}) \ \wedge \\ (m \in \mathrm{dom}\, \boldsymbol{e_1} \cap \mathrm{dom}\, \boldsymbol{e_2} \Rightarrow \boldsymbol{e_1}(m) = \boldsymbol{e_2}(m)) \end{array} \right. & & \textit{induction}
\end{array}
$$

According to the translation scheme, $[\![p']\!] = [\![p_1 \,|\, p_2]\!] = [\![p_1]\!] \mid [\![p_2]\!]$. *By application of* σ, *we have:*

$$\boldsymbol{e}\sigma = \boldsymbol{e_1}\sigma \cup \boldsymbol{e_2}\sigma$$

Using the fact that $\mathrm{fv}([\![p]\!]) = \mathrm{fv}(p)\sigma$, *and by application of the above induction hypothesis, we obtain that, for all* $x \in \mathrm{fv}([\![p]\!]) \cap \mathrm{fv}([\![q]\!])$,

$$(x \in \mathrm{dom}\, \boldsymbol{e_1}\sigma \Leftrightarrow x \in \mathrm{dom}\, \boldsymbol{e_2}\sigma)$$

and also

$$(x \in \mathrm{dom}\, \boldsymbol{e_1}\sigma \cap \mathrm{dom}\, \boldsymbol{e_2}\sigma \Rightarrow \boldsymbol{e_1}\sigma(x) = \boldsymbol{e_2}\sigma(x))$$

Now, by definition of the rule **(and)**, *we obtain:*

$$
\begin{array}{lll}
(p \,|\, q) \xrightarrow{\boldsymbol{e}} p' \Leftrightarrow ([\![p]\!] \mid [\![q]\!]) \xrightarrow{\boldsymbol{e}\sigma} [\![p']\!] & & \textit{rule } \textbf{(and)} \\
\Leftrightarrow ([\![p \,|\, q]\!]) \xrightarrow{\boldsymbol{e}\sigma} [\![p']\!] & & \textit{translation scheme}
\end{array}
$$

□

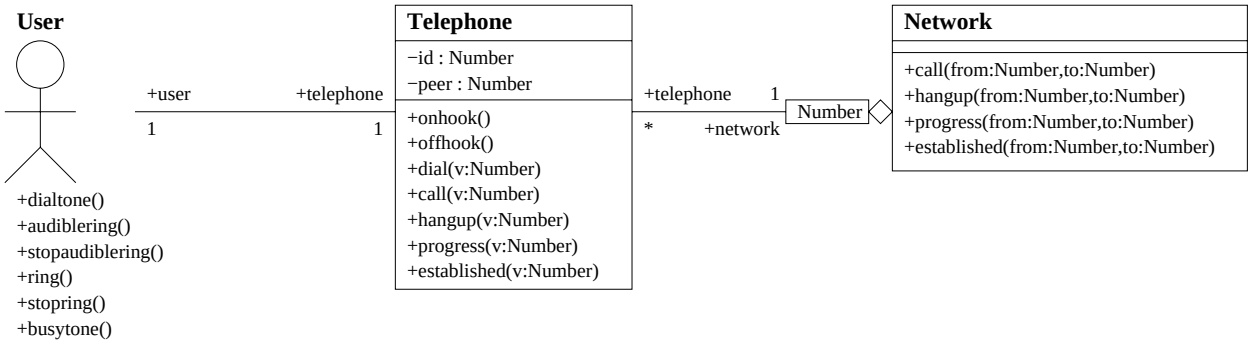


Fig. 1. Class diagram of the POTS

4 Case study: the Plain Old Telephone Service

This section aims at considering a real-world case-study that was pertinently suggested in [7] to advocate the benefits of object-oriented synchronous approach to the design of distributed reactive systems in UML. We use the UML specifications from [7] (figures 1, 3 and 2) to support our case study, showing the benefits of the notion of encapsulation and of the operator of behavioral inheritance provided by our synchronous model of computation.

4.1 Specification

4.1.1 The Plain Old Telephone Service

The UML class diagram of figure 1 depicts a simplified telecommunication system comprising terminals, subscriber-line management and network: a so-called "plain old telephone service" (POTS). The **Telephone** class is characterized by id, the number of the current instance, and by peer, the number of the called instance. Its methods implement the basic use of a telephone by a user (offhook, onhook and dial) and by the network (call, hangup, established and progress). The state diagram of figure 3 describes the behavior of a telephone. The two main sub-states of the diagram describe the behavior of a telephone when it is activated by

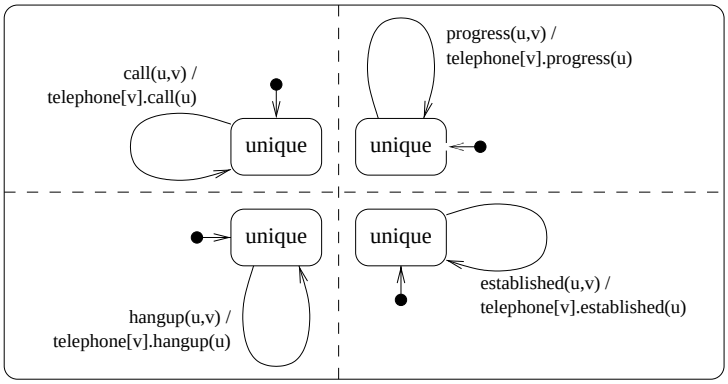


Fig. 2. State diagram of class **Network**

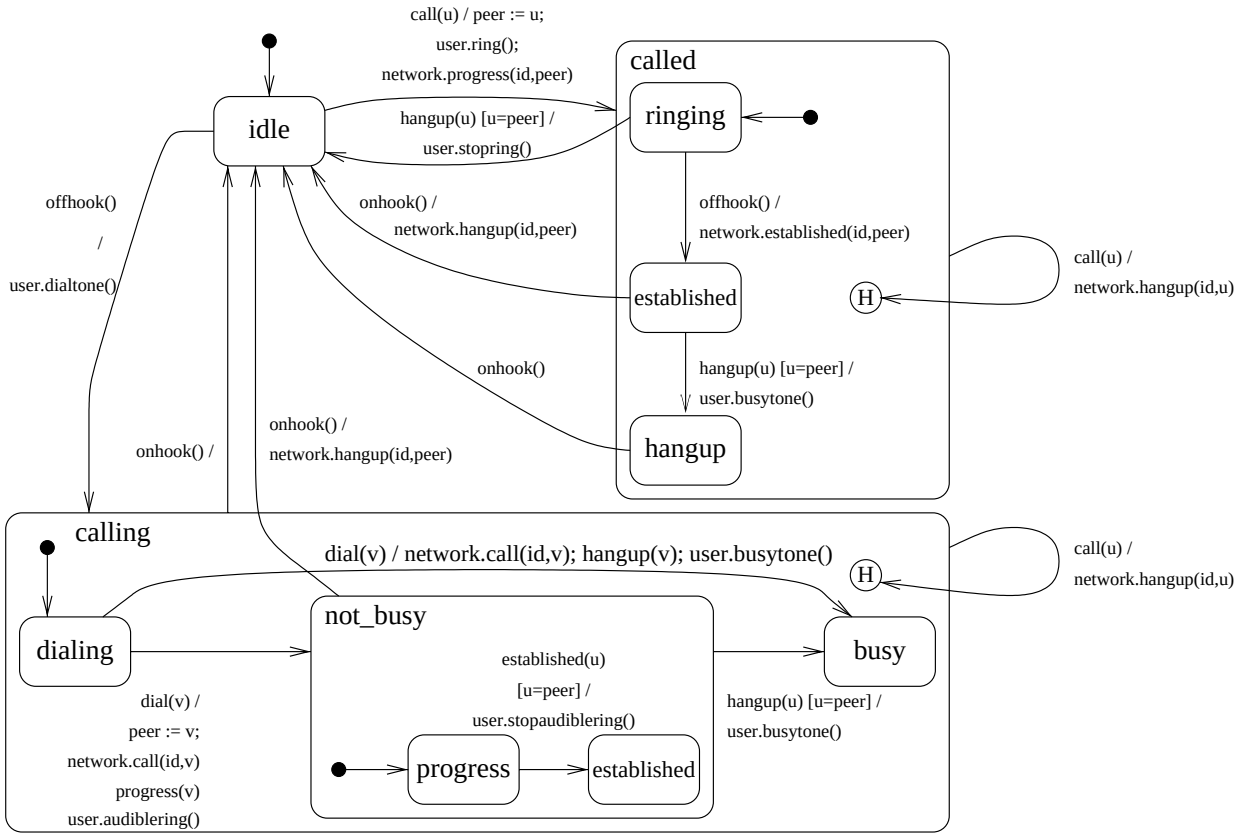


Fig. 3. State diagram of class **Telephone**

the network (state "called"), or by the user (state "calling"). In both cases, a new incoming call implies a hangup message to the caller. Also in both cases, the telephone is set back into its initial state ("idle") when the user hangs up (onhook). The state diagram of figure 2 describes the behavior of a basic network. An incoming message (call, hangup, established and progress) of a telephone identified by "u" is simply routed toward the corresponding telephone identified by "v".

4.1.2 Service adaptation: call forward on busy

Suppose that we wish to upgrade the POTS with a forwarding service ("call forward on busy"). It enables to forward an incoming call to a given number when the phone is busy instead of just returning a hangup message to the caller. Such a service adaptation implies the modification of **Telephone** and **Network**.

In the remainder of this section, we describe the expected modifications of the initial specification (depicted in figures 4, 5 and 6), and then we show how the inheritance operator achieves these modifications without breaking encapsulation.

There is a new attribute called fwd in **Telephone**. It contains the predefined number to which a call has to be forwarded when the user is busy. New methods forward in **Telephone**

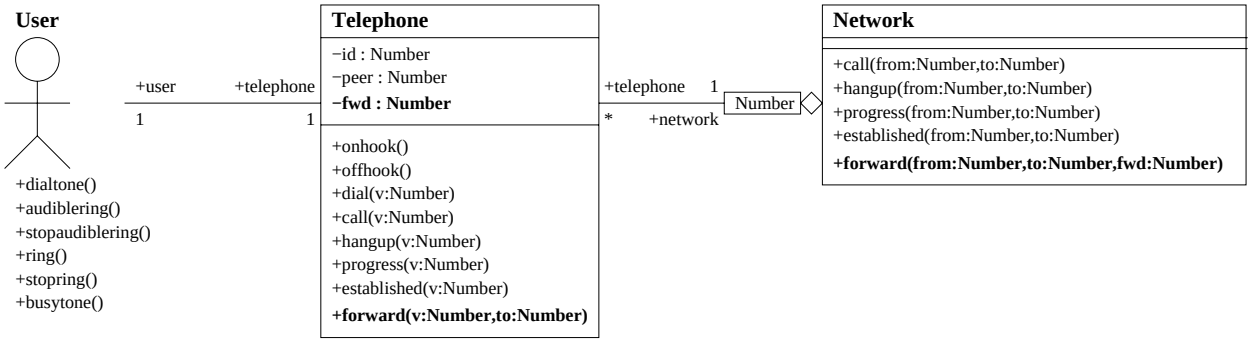


Fig. 4. Class diagram of the modified POTS

and **Network** implement the "call forward on busy" service. In **Telephone**, when the user dials a number, there is now a third state transition (toward "forwarded"). It corresponds to the answer forward(v, w) of the network. In this case the telephone dials itself the number w. For that purpose, it uses the attribute selfdial (in state "forwarded") exactly as the parameter v of call(v) in state "dialing". The class **Network** is also modified to take into account a new incoming message forward. This message is a request from "u" for forwarding the caller "v" to "f". This request is routed by the network toward the corresponding telephone "v". The figures 4, 5 and 6 show these modifications. In figure 5, the highlighted parts in gray point out the modifications. Two hangup messages have been preserved (boxed labels) when the user hangs up (onhook).

4.2 Synchronous adaptation

We obviously wish to achieve the service adaptation without breaking encapsulation, i.e. without rewriting **Network** and **Telephone**. We investigate the synchronous adaptation of **Telephone** using the inheritance mechanism shown in the previous section. First, we briefly describe the translation of a Statechart to an encapsulated Objective Signal process.

4.2.1 Statecharts encoding

For more readability, we do not take into account the "AND" super-states. Thus, a reaction of the system corresponds to a *unique* transition within the global Statechart. The behavior specified by a Statechart can then be described in Objective Signal by a choice between several transitions.

Current state The signals state and last_state carry the name of the current state and the name of the last state. The *clock* of these synchronized signals defines the rate of the whole process. We only consider the name of the *current activated basic* states (e.g. "progress") and not the nesting super states (e.g. "not_busy" or "calling"). This is allowed since only one basic state can be activated at a time (in the absence of "AND" super-states). Thus,

state is present at each transition, and last_state is defined as follows (i stands for the initial

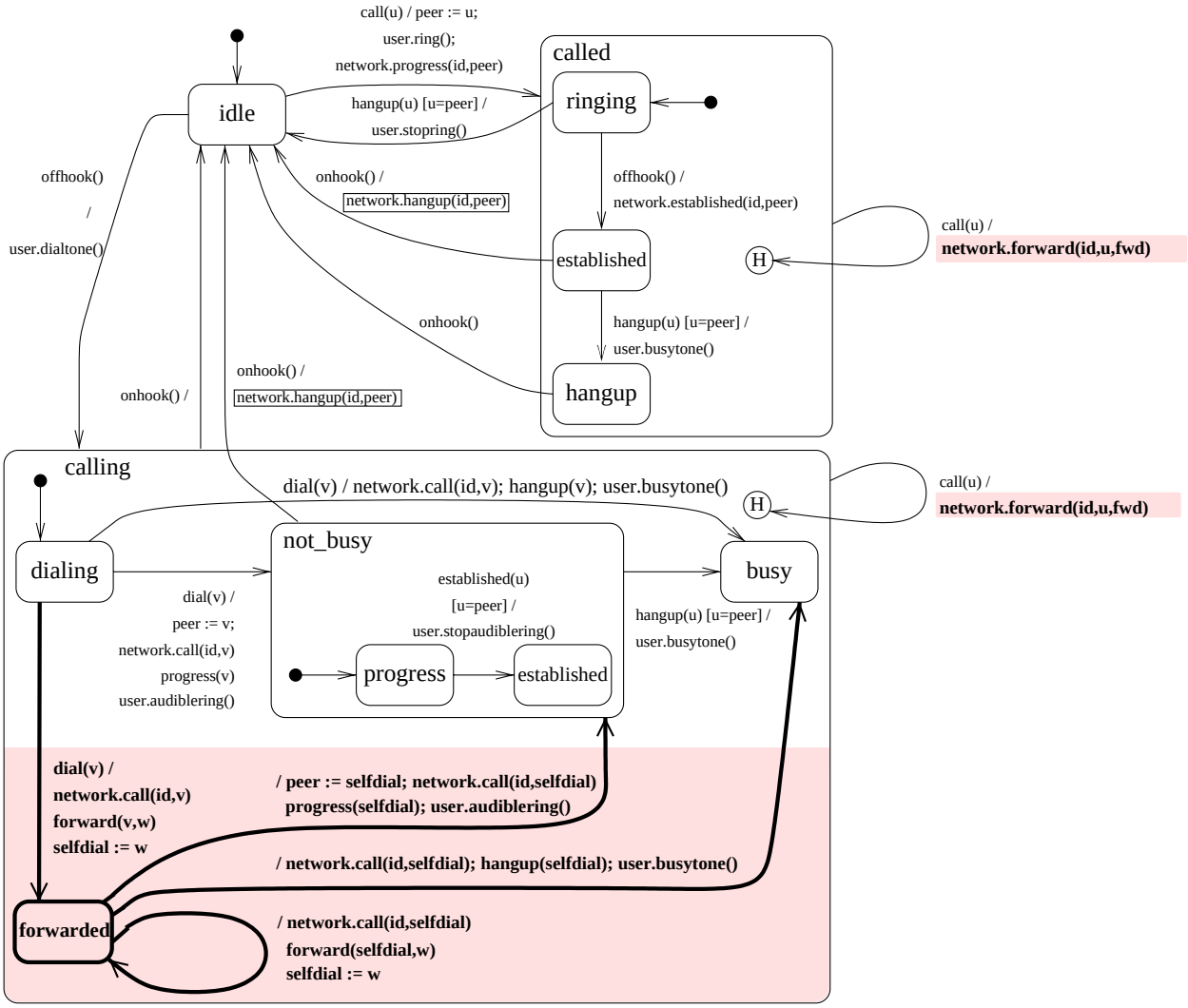


Fig. 5. State diagram of the modified class **Telephone**

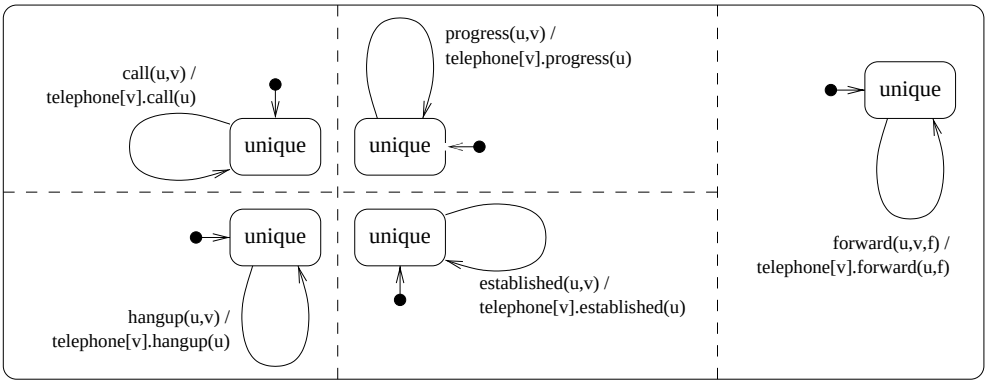


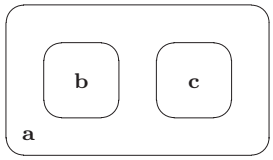
Fig. 6. State diagram of the modified class **Network**

state: idle in our example):

$$\mathsf{last\_state}{=}(\mathsf{pre}\ i)\ \mathsf{self.state}$$

Activated states For each state **a** (base state or not), we introduce a signal from_a that is synchronous with state. It indicates whether **a** is activated (or contains an activated sub-state) just before a transition. In the case of a basic state, we just have to check last_state. In the case of a super-state, we use a logical "OR" between the sub-states:

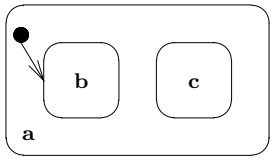
from_a = (self.last_state = a)



from_a = (self.from_b ∨ self.from_c)

Initial states For each state **a**, we introduce a signal init_a that is synchronous with state. It carries the name of the starting sub-states:

init_a = a

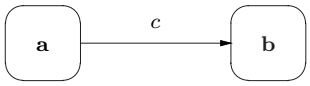


init_a = self.init_b

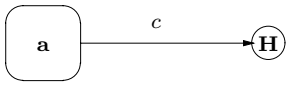
Historic The transition to the special (history) state H is encoded by a signal h_a that is synchronous with state, for each state **a**. If **a** is activated, then h_a is updated by last_state. Otherwise, the previous value of h_a is kept unchanged thanks by using a delayed version zh_a of h_a.

$$\left((\mathsf{zh\_a}{=}(\mathsf{pre\ a})\ \mathsf{self.h\_a}) \,\middle|\, \left(\begin{array}{c} (\mathsf{h\_a}{=}\mathsf{self.last\_state}) \,|\, (\mathsf{when\ self.from\_a}) \\ +\ (\mathsf{h\_a}{=}\mathsf{self.zh\_a}) \,|\, (\mathsf{when}(\mathsf{not\ self.from\_a})) \end{array} \right) \right) / \mathsf{zh\_a}$$

Transitions A transition labeled by c from **a** to **b** corresponds to an update of state. If the target of the transition is the history state H, the new state is h_b. This transition is guarded by $[\![c]\!]$, the encoding of the label c.



(when self.from_a) | $[\![c]\!]$ | (state = self.init_b)



(when self.from_a) | $[\![c]\!]$ | (state = self.h_b)

Labels of transitions The labels of the transitions correspond to synchronous constraints (signal calls) and to definitions of local signals.

$$[\![c/c']\!] = [\![c; c']\!] = [\![c]\!] \mid [\![c']\!]$$

The definition of a variable is encoded by the definition of a local signal, and the call of a method is encoded by the definition of its parameters. The translation of guards is straightforward.

$$[\![x := f(\tilde{y})]\!] = (x{=}f\ \tilde{y}) \qquad [\![m(\tilde{y})]\!] = (\tilde{y}{=}m) \qquad [\![[\ \mathsf{condition}\ ]]\!] = (\mathsf{when\ condition})$$

4.2.2 Call forward on busy

We focus on the translation of **Telephone** and its upgrade via inheritance. Assume that p_{tel} is the abstract behavior of the **Telephone** built using the translation scheme previously presented. Then, the class **Telephone** has the following interface:

$$\mathbf{Telephone} \stackrel{\mathsf{def}}{=} [\mathsf{user}, \mathsf{net}].[p_{\mathsf{tel}}]$$

The parameters user and net stand for the instance of **User** and **Network** required by the class diagram of figure 1.

Now, let us define the wrapper **CallForward** of **Telephone** to implement service adaptation without breaking encapsulation. We need to introduce the new state "forwarded". Thus, the signal state needs to be overridden. As in the example of the balance, the signal last_state now maintains a delayed version of *the new version* of state. The wrapper is a choice between the initial behavior and the new transitions to and from the state "forwarded". In the following definition of **CallForward**, we only make the transition from "dialing" to "forwarded"

explicit. The other transitions have a similar encoding:

$$\textbf{CallForward} \stackrel{\text{def}}{=} [\mathsf{user},\mathsf{net}].[].\left[\begin{array}{l} \big(\mathsf{state}{=}\mathsf{super.state}\big) \\ \left(\begin{array}{l} \big((x{=}(\mathsf{super.last\_state} = \mathit{dialing}))\,|\,(\mathsf{when\ self}.x)\big)/x \\ \mid \mathsf{state}{=}\mathit{forwarded} \\ \mid v{=}\mathsf{self.dial} \\ \mid (\mathsf{id}, v){=}\mathsf{net.call} \\ \mid (u, w){=}\mathsf{self.forward} \\ \mid \mathsf{selfdial}{=}\mathsf{self}.w \end{array}\right) \\ + \ldots \end{array}\right]$$

This wrapper overrides the signal state. The first branch of the choice allows to *reuse* the previous behavior of **Telephone**, *only when* the signals introduced in the other choices of the wrapper, namely forward, are *absent*. When forward is present, the new definition of state prevails (preemption). In this case, the last state must be "dialing" and the new state is "forwarded". The interface of **CallForward** holds the parameters user and net, which correspond to the same parameters as in the **Telephone**. Finally, the modification of **Telephone** by **CallForward** is simply achieved as follows:

$$\textbf{NewTelephone} \stackrel{\text{def}}{=} \textbf{Telephone}\&\textbf{CallForward}$$

5 Conclusion

There has been a lot of work aiming at combining object orientation and concurrency, especially within the framework of practical object-oriented languages like Eiffel [15,2]. However, our work aims at combining the notions of encapsulation and of inheritance found in object-oriented programming to a concurrency model that supports formal methods for modeling, verification and valid code generation purposes. To this end, many different approaches have already been considered for introducing object-oriented concepts in algebraic models of concurrency . In [18], Pierce and Turner introduce a simple object-based programming style in Pict, an implementation of the π-calculus. In Pict, objects are units of concurrency composed of several communicating agents. Objects in Pict aim at structuring a system, like modules (yet without functors). They encompass mobility but do not implement inheritance or overriding. A type-based notion of behavioral refinement of processes in Pict is obtained by superimposing a type inference system with subtyping to the process calculus. In [9], Fournet et al. propose an object-oriented extension of the join-calculus. A class-based inheritance mechanism is introduced that avoids the inheritance anomaly. In [5,6], Boussinot and Laneve

introduce an object-based *reactive* language. A notion of global *instants* is introduced. The same method cannot be executed more than once in the same instant. It is possible to clone an object, to add a method to an object and to rename a method. These operations implement a "derivation" mechanism rather than a "behavioral inheritance" mechanism stricto senso. For instance, if a modified object uses a currently overridden method, it still *uses* the previous version of this method.

In conclusion, we have introduced a new paradigm to express the essence of encapsulation and inheritance in a *synchronous* concurrent modeling framework. We introduced ISTS and Objective Signal, an extended version of ISTS build upon this paradigm. Reusability of components is achieved by an encapsulation mechanism. A static interpretation of inheritance allows a natural formulation of behavior refinement where overriding is taken into account. The proposed model supports a compile-time resolution of inheritance: overriding is interpreted statically and the inheritance combinator is translated into a primitive synchronous constructs. The benefits of our approach are illustrated by the adaptation of services to a plain old telephone service specification.

We implemented a prototype compiler from Objective Signal to SIGNAL based on the translation schemes from Objective Signal to ISTS (presented in this paper) and from ISTS to SIGNAL (presented in [14]). The use of enhanced features of SIGNAL (like modules to implement encapsulation), and the direct production of executable code from an executable ISTS specification are promising prospects.

References

[1] M. ABADI AND L. CARDELLI A Theory of Objects *Springer-Verlag* 1996.

[2] C. ATKINSON. Object-Oriented Reuse, Concurrency and Distribution *ACM Press and Addison-Wesley*, 1991.

[3] A. BENVENISTE, P. LE GUERNIC, C. JACQUEMOT. Synchronous programming with events and relations: the SIGNAL language and its semantics. In *Science of Computer Programming*, v. 16, 1991.

[4] G. BERRY, G. GONTHIER. The ESTEREL synchronous programming language: design, semantics, implementation. In *Science of Computer Programming*, v. 19, 1992.

[5] F. BOUSSINOT, C. LANEVE. Two Semantics for a Language of Reactive Objects *Research report n. 2511*. INRIA, March 1995.

[6] F. BOUSSINOT, G. DOUMENC, J.-M. STEFANI. Reactive Objects *Research report n. 2664*. INRIA, October 1995.

[7] B. CAILLAUD, J.-P. TALPIN, J.-M. JÉZÉQUEL, A. BENVENISTE AND C. JARD BDL: A Semantics Backbone for UML Dynamic Diagrams *Irisa/INRIA Rennes, research report RR-4003* 2000

[8] C. Flanagan and M. Abadi Object Types against Races *Proceedings International Conference on Concurrency Theory (CONCUR 99)* 1999.

[9] C. Fournet, L. Maranget, C. Laneve, D. Rémy. Inheritance in the Join Calculus. In *Foundations of Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science v. 1974. Springer, 2000.

[10] C. Fournet and G. Gonthier. The Reflexive CHAM and the Join-Calculus*Proceedings of the 23rd ACM Symposium on Principles of Programming Languages* 1996.

[11] A. D. Gordon and P. D. Hankin A Concurrent Object Calculus: Reduction and Typing *Proceedings High-Level Concurrent Languages (HLCL 98)* 1998.

[12] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud. The synchronous data-flow programming language Lustre. In *Proceedings of the* Ieee, v. 79(9). Ieee, 1991.

[13] D. Harel. Statecharts: a visual formalism for complex systems. In *Science of Computer Programming*, v. 8, 1987.

[14] M. Kerbœuf. Orientation objet d'un calcul de processus synchrones. *Thèse de doctorat*. Universit de Rennes 1, December 2002.

[15] B. Meyer. Systematic Concurrent Object-Oriented Programming *ISE, TR-EI-37/SC"* 1993.

[16] R. Milner. The Polyadic π-Calculus: A Tutorial *Logic and Algebra of Specification, Proceedings of International NATO Summer School* 1991.

[17] R. Milner. Communicating and Concurrency *Prentice-Hall International* 1989.

[18] B.C. Pierce, D.N. Turner Concurrent Objects in a Process Calculus In *Proceedings Theory and Practice of Parallel Programming (TPPP 94)* pp. 187-215. Takayasu Ito and Akinori Yonezawa, 1995.

[19] Pnueli, A., Shankar, N., Singerman, E. Fair synchronous transition systems and their liveness proofs. In *International School and Symposium on Formal Techniques in Real-time and Fault-tolerant Systems*. Lecture Notes in Computer Science v. 1468. Springer Verlag, 1998.

[20] B. C. Pierce and D. N. Turner Concurrent Objects in a Process Calculus *Proceedings Theory and Practice of Parallel Programming (TPPP 94)* 1994.

[21] J.-P. Talpin. Model checking robustness to desynchronization. In *Distributed and parallel embedded systems, IFIP World Computer Congress*. Kluwer Academic Publishers, August 2002.

A Implicit synchronous transition systems

A.1 Syntax

A.1.1 Kernel

$$
\begin{array}{lll}
p,q ::= \tilde{y}{=}f\tilde{x} & \text{(equation)} & \qquad c,d \in \mathcal{C} = \mathbb{B}+\mathbb{Z} \qquad \text{(constant)} \\
\quad | \quad y{=}(\mathsf{pre}\, c)x & \text{(transition)} & \qquad f,g \in \mathcal{F} \qquad \text{(function)} \\
\quad | \quad p \,|\, q & \text{(composition)} & \qquad x,y \in \mathcal{X} \qquad \text{(signal)} \\
\quad | \quad p{+}q & \text{(choice)} & \qquad (x_1,\ldots,x_k) \in \mathcal{X}^k \qquad \text{(sequence)} \\
\quad | \quad p/x & \text{(restriction)} & \qquad \tilde{x} \in \mathcal{X}^* = \bigcup_{k\in\mathbb{N}} \left(\mathcal{X}^k\right)
\end{array}
$$

A.1.2 Derived processes

$$
\begin{array}{rlcll}
\text{(guards)} & (\mathsf{when}\ x) & \stackrel{\text{def}}{=} & ((){=}\mathsf{when}\ x) & \\
 & (\mathsf{when}(\mathsf{not}\ x)) & \stackrel{\text{def}}{=} & ((){=}\mathsf{when}(\mathsf{not}\ x)) & \\
 & (\mathsf{event}\ x) & \stackrel{\text{def}}{=} & ((){=}\mathsf{event}\ x) & \\
\text{(constants)} & (x{=}\mathit{tt}) & \stackrel{\text{def}}{=} & (x{=}\mathsf{true}\ ()) & \\
 & (x{=}\mathit{ff}) & \stackrel{\text{def}}{=} & (x{=}\mathsf{false}\ ()) & \\
 & (x{=}n) & \stackrel{\text{def}}{=} & (x{=}\mathsf{n}\ ()) & (\forall n \in \mathbb{Z}) \\
\text{(silence)} & \mathsf{1} & \stackrel{\text{def}}{=} & ((){=}\mathsf{1}\ ()) &
\end{array}
$$

A.2 Algebraic laws

$$
\begin{array}{lll}
\mathrm{fv}(\tilde{y}=f\tilde{x}) = \tilde{y}\cup\tilde{x} & \mathrm{fv}(p{+}q) = \mathrm{fv}(p\,|\,q) = \mathrm{fv}(p)\cup\mathrm{fv}(q) & \mathrm{fv}(p/x) = \mathrm{fv}(p)\setminus\{x\} \\
\mathrm{dv}(\tilde{y}=f\tilde{x}) = \tilde{y} & \mathrm{dv}(p{+}q) = \mathrm{dv}(p\,|\,q) = \mathrm{dv}(p)\cup\mathrm{dv}(q) & \mathrm{dv}(p/x) = \mathrm{dv}(p)\setminus\{x\}
\end{array}
$$

$$
\begin{array}{llll}
p/y{\equiv}(p[x/y])/x^{(*)} & p\,|\,(q\,|\,r){\equiv}(p\,|\,q)\,|\,r & p{+}q{\equiv}q{+}p & p/x{\equiv}p^{(*)} \\
p/x/y{\equiv}p/y/x & p{+}(q{+}r){\equiv}(p{+}q){+}r & p\,|\,q{\equiv}q\,|\,p & \\
p\,|\,q/x{\equiv}(p\,|\,q)/x^{(*)} & p{+}q/x{\equiv}(p{+}q)/x^{(*)} & p\,|\,\mathsf{1}{\equiv}p{+}p\equiv p &
\end{array}
$$

A.3 Operationnal semantics

A.3.1 Environment and predefined functions

$$\begin{aligned} \boldsymbol{e}, \boldsymbol{f} \in \mathcal{E} = \mathcal{X} &\rightharpoonup \mathcal{C}^* && \text{(environment)} \\ \delta : \mathcal{F} &\rightharpoonup (\mathcal{C}^* \rightharpoonup \mathcal{C}^*) && \text{(predefined functions)} \\ \forall f \in \operatorname{dom}\delta, \quad \exists k, k' \text{ s.t.} \quad \delta(f) &: \mathcal{C}^k \rightharpoonup \mathcal{C}^{k'} \end{aligned}$$

$$\begin{aligned} \delta(+) &= \{((c,c') \mapsto (d)) \,|\, (c,c',d) \in \mathbb{Z}^3 \wedge d{=}c{+}c'\} & \delta(\mathsf{when}) &= \{((\#t) \mapsto ())\} \\ \delta(\mathsf{id}) &= \{((c) \mapsto (c)) | c \in \mathcal{C}\} & \delta(\mathsf{true}) &= \{(() \mapsto (\#t))\} \\ \delta(=) &= \{((c,c) \mapsto (\#t)) | c \in \mathcal{C}\} \cup \{((c,c') \mapsto (\#f)) | c \neq c'\} & \delta(\mathsf{false}) &= \{(() \mapsto (\#f))\} \\ \delta(\mathsf{event}) &= \{((c) \mapsto ()) \,|\, c \in \mathcal{C}\} & \delta(1) &= \{(() \mapsto ())\} \end{aligned}$$

A.3.2 Rules and axioms

$$(\text{eqv}) \frac{p \equiv p' \xrightarrow{\boldsymbol{e}} q' \equiv q}{p \xrightarrow{\boldsymbol{e}} q} \qquad (\text{or}) \frac{p \xrightarrow{\boldsymbol{e}} r}{p{+}q \xrightarrow{\boldsymbol{e}} r{+}q} \qquad (\text{let}) \frac{p \xrightarrow{\boldsymbol{e}} q}{p/x \xrightarrow{\boldsymbol{e}_x} q/x}$$

$$(\text{and}) \frac{p \xrightarrow{\boldsymbol{e}} p' \qquad q \xrightarrow{\boldsymbol{f}} q'}{p \,|\, q \xrightarrow{\boldsymbol{e} \cup \boldsymbol{f}} p' \,|\, q'} \quad \text{iff } \forall x \in \operatorname{fv}(p) \cap \operatorname{fv}(q), \begin{cases} (x \in \operatorname{dom}\boldsymbol{e} \Leftrightarrow x \in \operatorname{dom}\boldsymbol{f}) \\ \wedge\ (x \in \operatorname{dom}\boldsymbol{e} \cap \operatorname{dom}\boldsymbol{f} \Rightarrow \boldsymbol{e}(x) = \boldsymbol{f}(x)) \end{cases}$$

$$(\text{com}) \quad (\tilde{y}{=}f\tilde{x}) \xrightarrow{\boldsymbol{e}} (\tilde{y}{=}f\tilde{x}) \quad \text{iff } \operatorname{dom}\boldsymbol{e} = \tilde{x} \cup \tilde{y} \ \wedge\ \delta(f)(\boldsymbol{e}(\tilde{x})) = \boldsymbol{e}(\tilde{y})$$

$$(\text{pre}) \quad (y{=}(\mathsf{pre}\, c)x) \xrightarrow{x \mapsto d,\, y \mapsto c} (y{=}(\mathsf{pre}\, d)x)$$

A.3.3 Axioms derived from (com)

$$(z{=}x{+}y) \xrightarrow[1 \xrightarrow{\emptyset} 1]{x \mapsto c,\, y \mapsto d \quad z \mapsto c{+}d} (z{=}x{+}y) \qquad x{=}tt \xrightarrow{x \mapsto tt} x{=}tt \qquad \mathsf{when}\ x \xrightarrow{x \mapsto tt} \mathsf{when}\ x$$

$$x{=}ff \xrightarrow{x \mapsto ff} x{=}ff \qquad \mathsf{event}\ x \xrightarrow{x \mapsto c} \mathsf{event}\ x$$

B Objective Signal

B.1 Syntax

$$\begin{array}{rcll}
p, q & ::= & p \,|\, q \mid p{+}q \mid p/m \mid \tilde{m}{=}f\tilde{m}' \mid m{=}(\mathsf{pre}c)m' \mid o' = \mathsf{new}\ \mathbb{C}(\tilde{o}) & \text{(instanciated process)} \\
m & ::= & o.x & \text{(instanciated path)} \\
o & \in & \mathcal{O} & \text{(object name)} \\
\mathbb{C} & ::= & [\tilde{C_r}].[\tilde{C}].[p_a] \quad | \quad \mathbb{C}\&\mathbb{C}' & \text{(class)} \\
p_a, q_a & ::= & p_a \,|\, q_a \mid p_a{+}q_a \mid p_a/x \mid \tilde{y}{=}f\tilde{n} \mid y{=}(\mathsf{pre}c)n & \text{(abstract process)} \\
n & ::= & C.x & \text{(abstract path)} \\
C & \in & \mathcal{M} \supset \{\mathsf{super}, \mathsf{self}\} & \text{(class parameter)}
\end{array}$$

B.2 Operationnal semantics

B.2.1 Renaming operators

$$\mathsf{bind}^o_\sigma(p_a \,|\, q_a) = \mathsf{bind}^o_\sigma(p_a) \,|\, \mathsf{bind}^o_\sigma(q_a) \qquad \mathsf{bind}^o_\sigma(p_a/x) = (\mathsf{bind}^o_\sigma(p_a))/o.x$$

$$\mathsf{bind}^o_\sigma(p_a{+}q_a) = \mathsf{bind}^o_\sigma(p_a){+}\mathsf{bind}^o_\sigma(q_a) \qquad \mathsf{bind}^o_\sigma(\tilde{y}{=}f\tilde{n}) = (o.\tilde{y}{=}f(\tilde{n}\sigma))$$

$$\beta_{p_a q_a} : \begin{cases} \mathrm{dom}\,\beta_{p_a q_a} = \mathrm{dv}(p_a) \cap \mathrm{dv}(q_a) \\ \forall x \in \mathrm{dom}\,\beta_{p_a q_a},\ x\beta_{p_a q_a} = x' \notin \mathrm{dv}(p_a) \cup \mathrm{dv}(q_a) \end{cases}$$

$$\mathcal{L}_\beta\,(\,p_a \,|\, q_a\,) = \mathcal{L}_\beta(p_a)\,|\,\mathcal{L}_\beta(q_a) \qquad \mathcal{L}_\beta(p_a/x) = \mathcal{L}_{\beta_x}\,(p_a)\,/x$$
$$\mathcal{L}_\beta\,(\,p_a{+}q_a\,) = \mathcal{L}_\beta(p_a){+}\mathcal{L}_\beta(q_a) \qquad \mathcal{L}_\beta(\tilde{y}{=}f\tilde{n}) = \Big((\tilde{y}\beta){=}f\tilde{n}\Big)$$

$$\begin{aligned}&\mathsf{lookup}\left(\begin{array}{c}[\tilde{C_{r1}}]\,.\,[\tilde{C_1}]\,.\,[p_a]\\ \&\;[\tilde{C_{r2}}]\,.\,[\tilde{C_2}]\,.\,[q_a]\end{array}\right)\\ &\quad= [\tilde{C_{r1}}].[\tilde{C_1}\tilde{C_2}]\,.\\ &\qquad\left[\left(\begin{array}{l}\quad\mathcal{L}_{\beta_{p_aq_a}}(p_a)\\ \mid\; q_a[\tilde{C_{r1}}\tilde{C_1}/\tilde{C_{r2}}][\mathsf{self}.(x\beta_{p_aq_a})/\mathsf{super}.x]\end{array}\right)/\mathrm{im}\,\beta_{p_aq_a}\right]\end{aligned}$$

$$\mathsf{lookup}\,(\,\mathbb{C}\&\mathbb{C}'\,) = \mathsf{lookup}\,(\,\mathsf{lookup}(\mathbb{C})\&\mathsf{lookup}(\mathbb{C}')\,)$$
$$\mathsf{lookup}(\mathbb{C}) = \mathbb{C} \qquad (\text{with } \mathbb{C}\text{: base class})$$

B.2.2 Rule for class instanciation

$$(\text{inst})\quad \frac{\mathsf{lookup}(\mathbb{C}) = [\tilde{C}].[p_a] \qquad\qquad \mathsf{bind}^{o'}_{[o'\tilde{o}/\mathsf{self}\tilde{C}]}(p_a) \xrightarrow{\boldsymbol{e}} p}{o' = \mathsf{new}\ \mathbb{C}(\tilde{o}) \xrightarrow{\boldsymbol{e}} p}$$

B.3 From Objective Signal to ISTS

$$[\![o' = \mathsf{new}\ \mathbb{C}(\tilde{o})]\!] = [\![o' = \mathsf{new}\ \mathsf{lookup}(\mathbb{C})(\tilde{o})]\!] \qquad [\![p\,|\,q]\!] = [\![p]\!]\ |\ [\![q]\!]$$
$$\Big[\!\Big[o' = \mathsf{new}\ [\tilde{C}].[p_a](\tilde{o})\Big]\!\Big] = \Big[\!\Big[\mathsf{bind}^{o'}_{[o'\tilde{o}/\mathsf{self}\tilde{C}]}(p_a)\Big]\!\Big] \qquad [\![p{+}q]\!] = [\![p]\!] + [\![q]\!]$$
$$[\![\tilde{m} = f\tilde{m}']\!] = (\tilde{m}\sigma = f\tilde{m}'\sigma) \qquad [\![p/m]\!] = [\![p]\!]\,/(m\sigma)$$

Contents